Ejercicio Guiado Tool SECURITYP1

Andres Valdivieso Pinilla

Me pareció adecuado crear **un ejercicio guiado, estructurado y reproducible**, pensado en demostrar **todas** las funcionalidades de esta herramienta en Windows (PowerShell) y Kali/Linux (bash), **sin usar EICAR ni ejecutar malware**.

Cada paso incluye: comando, qué hace, salida esperada y cómo verificar/generar reportes. Al final tendran dos scripts (PowerShell + Bash) que ejecutan todo automáticamente y guardan resultados en `reports/`.

Supuestos:

> - Están en la raíz del repo (donde está `main.py`)
> - Estructura: `modules/`, `reports/`, `scripts/`, `test_data/`.
> - `test_data/` contiene: `notepad_sample.exe` (o placeholder `MZ`), `random.bin`, `wordlist.txt`.
> - No vamos a subir archivos que triggerneen AV real. Usaremos placeholders y binarios inofensivos.

Índice

Disclamer

1. El siguiente ejercicio y los scripts asociados se proporcionan únicamente con fines educativos y para comprender el funcionamiento y las capacidades de la herramienta.
2. Las pruebas deben realizarse en entornos controlados (máquinas de laboratorio, contenedores o imágenes aisladas) y con copias de seguridad; ejecutar comandos en entornos productivos o sin autorización puede causar pérdida de datos o interrupciones.
3. El autor/propietario del repositorio no se hace responsable de daños, pérdidas o perjuicios derivadas del uso indebido o de la ejecución de los comandos fuera de un entorno controlado. Cada usuario es responsable de obtener las autorizaciones necesarias y de aplicar las precauciones apropiadas antes de ejecutar las pruebas.

---

0 - Preparación inicial (ambas plataformas)

1. Abre shell en la carpeta raíz del proyecto.
2. Crea `reports/` si no existe:

- PowerShell:

```powershell
if (-not (Test-Path .\reports)) { New-Item -ItemType Directory -Path .\reports | Out-Null }
```

- Bash:

```bash
mkdir -p reports
```

3. Comprueba variables de entorno (VirusTotal / FileScan):

- PowerShell:

```powershell
# Temporales en esta sesión (no persisten)
$env:VT_API_KEY = "TU_VT_API_KEY_AQUI"
$env:FILESCAN_API_KEY = "TU_FILESCAN_API_KEY_AQUI"
$env:SHODAN_API_KEY="TU_SHODAN_API_KEY_AQUI"
echo $env:VT_API_KEY
echo $env:FILESCAN_API_KEY
echo $env:SHODAN_API_KEY
```

- Bash:

```bash
export VT_API_KEY="TU_VT_API_KEY_AQUI"
export FILESCAN_API_KEY="TU_FILESCAN_API_KEY_AQUI"
export SHODAN_API_KEY="TU_SHODAN_API_KEY_AQUI"
echo $VT_API_KEY
```

```
echo $FILESCAN_API_KEY
echo $SHODAN_API_KEY
```

Si no van a probar VT/FileScan/Shodan por falta de las llaves, se pueden dejar vacíos — los pasos saltarán o devolverán error controlado (esperado).

4. Instala dependencias si hace falta (Kali):

```
sudo apt update
sudo apt install -y python3 python3-pip clamav binutils strace ltrace ss net-tools
python3 -m pip install -r requirements.txt
```

En Windows instala sólo Python + dependencias de `requirements.txt`. Herramientas tipo `strings.exe` vienen con Sysinternals si las quieres usar.

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

---

1 — Generar datos de prueba (sin EICAR)

Usaremos un placeholder para el test antivirus en lugar de EICAR.

PowerShell:

```
$BaseDir = (Join-Path (Get-Location) "test_data")

New-Item -ItemType Directory -Path $BaseDir -Force | Out-Null



# Placeholder pequeño (NO EICAR)

'NOT-EICAR-TEST-PLACEHOLDER' | Out-File -FilePath (Join-Path $BaseDir 'eicar_placeholder.com') -
Encoding ASCII -NoNewline -Force



# wordlist

"password`n123456" | Out-File -FilePath (Join-Path $BaseDir 'wordlist.txt') -Encoding ASCII -Force



# random binary 10 KiB

$bytes = New-Object byte[] 10240

[System.Random]::new().NextBytes($bytes)

[System.IO.File]::WriteAllBytes((Join-Path $BaseDir 'random.bin'), $bytes)



# sample binary placeholder (copy notepad if exists; else tiny MZ)

$notepad = "$env:windir\System32\notepad.exe"

if (Test-Path $notepad) {

    Copy-Item -Path $notepad -Destination (Join-Path $BaseDir 'notepad_sample.exe') -Force

} else {

    'MZ' | Out-File -FilePath (Join-Path $BaseDir 'notepad_sample.exe') -Encoding ASCII -NoNewline -
Force
```

```
}


Get-ChildItem $BaseDir | Format-Table Name,Length
```

Bash:

```
mkdir -p test_data

printf 'NOT-EICAR-TEST-PLACEHOLDER' > test_data/eicar_placeholder.com

printf "password\n123456\n" > test_data/wordlist.txt

dd if=/dev/urandom of=test_data/random.bin bs=1024 count=10 2>/dev/null || head -c 10240 /dev/urandom >
test_data/random.bin

cp /usr/bin/true test_data/notepad_sample.exe 2>/dev/null || printf 'MZ' > test_data/notepad_sample.exe

ls -l test_data/
```

Explicación de los dos bloques

Preparación del directorio base

- `$BaseDir = (Join-Path (Get-Location) "test_data")`
  Define una variable `$BaseDir` que apunta a una subcarpeta llamada `test_data` dentro del directorio de trabajo actual. Usa `Join-Path` para construir la ruta de forma robusta.
- `New-Item -ItemType Directory -Path $BaseDir -Force | Out-Null`
  Crea el directorio `test_data` si no existe. El parámetro `-Force` permite crearlo aun cuando ya exista (evita error) y `Out-Null` descarta la salida para que no se muestre en pantalla

Creación de un placeholder pequeño (nota: NO EICAR)

- `# Placeholder pequeño (NO EICAR)` — comentario indicando intención de no generar el fichero de prueba EICAR real.
- `'NOT-EICAR-TEST-PLACEHOLDER' | Out-File -FilePath (Join-Path $BaseDir 'eicar_placeholder.com') -Encoding ASCII -NoNewline -Force`
  Escribe la cadena literal `NOT-EICAR-TEST-PLACEHOLDER` en un archivo llamado `eicar_placeholder.com` dentro de `test_data`.
  - `-Encoding ASCII` fuerza codificación ASCII.
  - `-NoNewline` evita añadir una nueva línea final.
  - `-Force` sobrescribe el archivo si ya existe.
    Objetivo: crear un fichero *placeholder* que imite la presencia de un archivo de prueba sin usar la firma EICAR real (útil para pruebas seguras).

Generación de una wordlist

- `# wordlist` — comentario aclaratorio.
- `"password` n123456" | Out-File -FilePath (Join-Path $BaseDir 'wordlist.txt') -Encoding ASCII -Force `Crea un archivo` wordlist.txt `con dos líneas (` password `y` 123456 `). El` `n` es una nueva línea literal en la cadena double-quoted para PowerShell. Se usa ASCII y `-Force` para sobrescribir si existe. Objetivo: disponer de una lista de contraseñas de ejemplo para pruebas (fuerza bruta, testeo de herramientas, etc.).

Generar datos binarios aleatorios (10 KiB)

- `# random binary 10 KiB` — comentario.
- `$bytes = New-Object byte[] 10240`
  Reserva un array de bytes de tamaño 10240 (10 KiB).

- `[System.Random]::new().NextBytes($bytes)`
  Rellena el array con bytes pseudoaleatorios.
- `[System.IO.File]::WriteAllBytes((Join-Path $BaseDir 'random.bin'), $bytes)`
  Escribe los bytes en `random.bin` dentro de `test_data`. Objetivo: crear un archivo binario de prueba con contenido no estructurado para validar detecciones, tamaños, lecturas binarias, etc.

Crear un sample binario (copia de notepad.exe si existe; si no, crear un pequeño MZ)

- `# sample binary placeholder (copy notepad if exists; else tiny MZ)` — comentario.
- `$notepad = "$env:windir\System32\notepad.exe"`
  Construye la ruta esperada a `notepad.exe` usando la variable de entorno `windir`.
- `if (Test-Path $notepad) { Copy-Item -Path $notepad -Destination (Join-Path $BaseDir 'notepad_sample.exe') -Force } else { 'MZ' | Out-File -FilePath (Join-Path $BaseDir 'notepad_sample.exe') -Encoding ASCII -NoNewline -Force }`
  Comprueba si `notepad.exe` existe en la ubicación esperada.
  - Si existe, copia `notepad.exe` a `test_data\notepad_sample.exe` (proporcionando un binario real como placeholder).
  - Si no existe, crea un archivo `notepad_sample.exe` que contiene solo los caracteres `MZ` (las dos letras iniciales del encabezado PE) — suficiente para simular un ejecutable muy simple.
    `-Force` sobrescribe la copia si existe. Objetivo: disponer de un archivo que aparenta ser un ejecutable para pruebas (análisis estático, heurísticas, motor EDR en entorno controlado), sin introducir software adicional.

Listar contenido del directorio

- `Get-ChildItem $BaseDir | Format-Table Name,Length`
  Enumera el contenido de `test_data` y lo formatea como tabla mostrando `Name` y `Length` (tamaño en bytes). Es la salida final que confirma los archivos creados y sus tamaños.

Consideraciones de seguridad y buenas prácticas

- El script evita explícitamente el uso de la cadena EICAR real al colocar un placeholder (`NOT-EICAR-TEST-PLACEHOLDER`). Es buena práctica para no disparar detecciones reales ni compartir muestras maliciosas.
- Ejecutar este script en un *entorno controlado* (máquina de pruebas o contenedor) es recomendable; aunque el script no descarga ni ejecuta malware, copiar ejecutables del sistema (como notepad.exe) puede no ser deseable en algunos contextos regulados.
- Si se quiere simular firmas reales para pruebas de detección, se debe coordinar con el equipo de seguridad y usar muestras aprobadas o entornos aislados.
- Los archivos creados con `-Force` sobrescribirán ficheros existentes con nombres idénticos; tener esto en cuenta para evitar pérdida de datos.

Resumen rápido (qué archivos crea)

- `test_data\eicar_placeholder.com` — texto: `NOT-EICAR-TEST-PLACEHOLDER` (sin newline).
- `test_data\wordlist.txt` — dos líneas: `password` y `123456`.
- `test_data\random.bin` — 10 KiB de datos binarios aleatorios.
- `test_data\notepad_sample.exe` — copia de Notepad si existe, o un archivo con `MZ` si no.
- Comando final muestra nombre y tamaño de estos ficheros.

Verifica: `test_data/` contiene `eicar_placeholder.com`, `notepad_sample.exe`, `random.bin`, `wordlist.txt`.

---

2 — Módulo *forensics* (hashes)

Objetivo: obtener md5/sha1/sha256 de un fichero para usarlo con VT.

PowerShell:

```powershell
$p = (Resolve-Path .\test_data\notepad_sample.exe).Path

python .\main.py forensics --file-hash "$p" | Tee-Object -FilePath reports/forensics_notepad.json
```

Bash:

```bash
p=$(realpath test_data/notepad_sample.exe)

python3 main.py forensics --file-hash "$p" | tee reports/forensics_notepad.json
```

Salida esperada:

- JSON o texto con `md5`, `sha1`, `sha256`.
- Verifica que `reports/forensics_notepad.json` exista y contenga esos valores.

Explicación de los dos bloques

- `$p = (Resolve-Path .\test_data\notepad_sample.exe).Path`
  Define la variable `$p` con la ruta absoluta al fichero `notepad_sample.exe` dentro del subdirectorio `test_data`. `Resolve-Path` convierte la ruta relativa en la ruta completa, y `.Path` extrae la cadena de ruta.
- `python .\main.py forensics --file-hash "$p"`
  Ejecuta el script Python `main.py` con el subcomando `forensics` y pasa el valor de la variable `$p` al parámetro `--file-hash`. Las comillas dobles (`"$p"`) permiten que PowerShell expanda la variable y preserven espacios en la ruta si las hubiera.
- `| Tee-Object -FilePath reports/forensics_notepad.json`
  La salida estándar del comando Python se canaliza (`|`) a `Tee-Object`. Esto hace que la salida se muestre en pantalla **y** se copie simultáneamente al archivo `reports/forensics_notepad.json`. Si la carpeta `reports` no existe, PowerShell devolverá un error al intentar escribir el archivo.

---

3 — Módulo *malware* — extractor Python (strings) y entropía

3.1 Strings (sin herramientas externas):

```powershell
python .\main.py malware --strings .\test_data\notepad_sample.exe | Tee-Object
reports/malware_strings_notepad.txt
```

o

```bash
python3 main.py malware --strings test_data/notepad_sample.exe | tee
reports/malware_strings_notepad.txt
```

Verifica: lista de strings (muestra) y que se haya guardado el archivo.

3.2 Entropía:

```powershell
python .\main.py malware --entropy .\test_data\random.bin | Tee-Object
reports/malware_entropy_random.txt
```

Explicación de los dos bloques

- `python .\main.py malware --strings .\test_data\notepad_sample.exe | Tee-Object reports/malware_strings_notepad.txt`
  Ejecuta `main.py` con el subcomando `malware` y el argumento `--strings` apuntando al archivo `.\test_data\notepad_sample.exe`. La salida estándar (presumiblemente la lista de *strings* extraídas) se duplica: se muestra en pantalla y se graba en `reports/malware_strings_notepad.txt` mediante `Tee-Object`.
- `python3 main.py malware --strings test_data/notepad_sample.exe | tee reports/malware_strings_notepad.txt`
  Ejecuta `main.py` con `python3` (equivalente al anterior en entornos Unix), pidiendo la extracción de *strings* de `test_data/notepad_sample.exe`. La salida se muestra en pantalla y se escribe en `reports/malware_strings_notepad.txt` usando `tee`.

Verifica: valor numérico; `random.bin` debería mostrar entropía alta (~7.5–8.0), placeholder `notepad_sample.exe` baja.

Verificación

PowerShell (Windows)

1. Comprobar que el archivo `reports/malware_strings_notepad.txt` existe:
   `Test-Path .\reports\malware_strings_notepad.txt`

- Resultado esperado: `True` (si existe).

2. Ver tamaño y metadatos:
   `Get-Item .\reports\malware_strings_notepad.txt | Select-Object Name,Length,LastWriteTime`

- Resultado esperado: fila con `Name`, `Length` (bytes > 0 si hay contenido) y `LastWriteTime`.

3. Mostrar una **muestra** (primeras 20 líneas) de la lista de strings:
   `Get-Content .\reports\malware_strings_notepad.txt -TotalCount 20`

- Resultado esperado: hasta 20 líneas de texto (cada línea es una *string* extraída).

4. (Opcional) Si quiere ver *strings* directamente del binario con la utilidad `strings` (si está disponible):
   `strings .\test_data\notepad_sample.exe | Select-Object -First 20`

Bash (Linux / macOS)

1. Comprobar existencia:
   `[ -f reports/malware_strings_notepad.txt ] && echo "exists" || echo "missing"`

- Resultado esperado: `exists`.

2. Ver tamaño y fecha:
   `ls -l reports/malware_strings_notepad.txt`

- Resultado esperado: línea con permisos, tamaño en bytes (>0 si tiene contenido) y fecha.

3. Mostrar una **muestra** (primeras 20 líneas):
   `head -n 20 reports/malware_strings_notepad.txt`

- Resultado esperado: hasta 20 líneas de texto con *strings*.

4. (Opcional) Extraer *strings* directamente del ejecutable usando `strings` y mostrar 20 primeras:
   `strings test_data/notepad_sample.exe | head -n 20`

Qué se Confirmaría visualmente

- Que `reports/malware_strings_notepad.txt` **existe** y tiene un `Length`/tamaño > 0.
- Que al desplegar las primeras líneas aparezcan **líneas legibles** (palabras, rutas, identificadores, mensajes), es decir, fragmentos imprimibles extraídos del binario — esto corresponde a la “lista de strings (muestra)”.

---

4 — Módulo *malware* — ejecutar herramienta puntual (`--tool`)

Usa `--tool` para ejecutar comandos concretos (objdump/xxd/strace/netstat) sin cambiar la interfaz principal.

Ejemplos:

PowerShell:

```
# strings por Sysinternals (Windows) o builtin

python .\main.py malware --tool sysstrings --target .\test_data\notepad_sample.exe | Tee-Object
reports/malware_tool_sysstrings.txt



# netstat (sin target)

python .\main.py malware --tool netstat | Tee-Object reports/malware_tool_netstat.txt
```

Bash:

```
python3 main.py malware --tool objdump --target test_data/notepad_sample.exe | tee
reports/malware_tool_objdump.txt

python3 main.py malware --tool ss | tee reports/malware_tool_ss.txt
```

Explicación de los dos bloques

PowerShell (Windows)

- `python .\main.py malware --tool sysstrings --target .\test_data\notepad_sample.exe | Tee-Object reports/malware_tool_sysstrings.txt`
  Ejecuta `main.py` con el subcomando `malware`, especificando la herramienta `sysstrings` y el archivo de destino `.\test_data\notepad_sample.exe`. El resultado de la ejecución se muestra en pantalla y, a la vez, se guarda en `reports/malware_tool_sysstrings.txt` mediante `Tee-Object`.
- `python .\main.py malware --tool netstat | Tee-Object reports/malware_tool_netstat.txt`
  Ejecuta `main.py` con el subcomando `malware`, indicando la herramienta `netstat` pero sin un archivo de destino. La salida se duplica: aparece en la consola y se escribe en `reports/malware_tool_netstat.txt`.

Bash (Linux / macOS)

- `python3 main.py malware --tool objdump --target test_data/notepad_sample.exe | tee reports/malware_tool_objdump.txt`
  Ejecuta `main.py` con el subcomando `malware`, usando la herramienta `objdump` sobre el archivo `test_data/notepad_sample.exe`. La salida del análisis se muestra en pantalla y se guarda en `reports/malware_tool_objdump.txt` gracias al comando `tee`.
- `python3 main.py malware --tool ss | tee reports/malware_tool_ss.txt`
  Ejecuta `main.py` con el subcomando `malware`, llamando la herramienta `ss` sin un archivo de destino. El resultado se imprime en pantalla y se guarda en `reports/malware_tool_ss.txt` mediante `tee`.

Verificación

PowerShell

1. Confirmar existencia de archivos de salida:
   `Test-Path .\reports\malware_tool_sysstrings.txt Test-Path .\reports\malware_tool_netstat.txt`
   Resultado esperado: `True` en ambos.
2. Revisar tamaño y fecha de modificación:
   `Get-Item .\reports\malware_tool_sysstrings.txt, .\reports\malware_tool_netstat.txt | Select-Object`

`Name,Length,LastWriteTime`

3. Ver una muestra de resultados (primeras 15 líneas):
   `Get-Content .\reports\malware_tool_sysstrings.txt -TotalCount 15 Get-Content .\reports\malware_tool_netstat.txt -TotalCount 15`

Bash

1. Confirmar existencia de archivos de salida:
   `[ -f reports/malware_tool_objdump.txt ] && echo "objdump ok" || echo "objdump missing" [ -f reports/malware_tool_ss.txt ] && echo "ss ok" || echo "ss missing"`
2. Revisar tamaño y fecha:
   `ls -lh reports/malware_tool_objdump.txt reports/malware_tool_ss.txt`
3. Ver una muestra de resultados (primeras 15 líneas):
   `head -n 15 reports/malware_tool_objdump.txt head -n 15 reports/malware_tool_ss.txt`

Notas:

- Si la herramienta no está instalada verás `executable not found` — esto también prueba la robustez del código.
- Los comandos cuyo output es muy grande se retornan parcialmente (`stdout_preview`).

---

5 — Módulo *malware* — *deep-scan* (batería)

Ejecuta la batería predefinida (no hace red, sólo herramientas locales; si faltan se saltan).

PowerShell:

```
python .\main.py malware --deep-scan .\test_data\notepad_sample.exe | Tee-Object reports/malware_deepscan_notepad.json
```

Bash:

```
python3 main.py malware --deep-scan test_data/notepad_sample.exe | tee reports/malware_deepscan_notepad.json
```

Explicación de los dos bloques

PowerShell (Windows)

- `python .\main.py malware --deep-scan .\test_data\notepad_sample.exe | Tee-Object reports/malware_deepscan_notepad.json`
  Ejecuta `main.py` con el subcomando `malware` y el argumento `--deep-scan` aplicado al archivo `.\test_data\notepad_sample.exe`. La salida resultante, en formato JSON, se muestra en consola y se guarda en `reports/malware_deepscan_notepad.json` mediante `Tee-Object`.

Bash (Linux / macOS)

- `python3 main.py malware --deep-scan test_data/notepad_sample.exe | tee reports/malware_deepscan_notepad.json`
  Ejecuta `main.py` con el subcomando `malware` y el argumento `--deep-scan` sobre el archivo `test_data/notepad_sample.exe`. La salida se imprime en pantalla y se almacena en `reports/malware_deepscan_notepad.json` con ayuda del comando `tee`.

Verificación

PowerShell

1. Comprobar existencia del archivo JSON:
   `Test-Path .\reports\malware_deepscan_notepad.json`
2. Ver tamaño y metadatos del archivo:
   `Get-Item .\reports\malware_deepscan_notepad.json | Select-Object Name,Length,LastWriteTime`
3. Mostrar una muestra de contenido (*primeras 20 líneas*):
   `Get-Content .\reports\malware_deepscan_notepad.json -TotalCount 20`

Bash

1. Confirmar existencia del archivo JSON:
   `[ -f reports/malware_deepscan_notepad.json ] && echo "exists" || echo "missing"`
2. Revisar tamaño y fecha:
   `ls -lh reports/malware_deepscan_notepad.json`
3. Mostrar una muestra de contenido (*primeras 20 líneas*):
   `head -n 20 reports/malware_deepscan_notepad.json`

---

6 — VirusTotal (upload y hash lookup) — opcional

Sólo si tienes `VT_API_KEY`. *No uses EICAR real*.

6.1 Subida:

```
python .\main.py --case CASE-001 malware --vt-upload .\test_data\notepad_sample.exe | Tee-Object
reports/vt_upload_notepad.json
```

6.2 Lookup por hash (usa SHA256 del paso forensics):

```
python .\main.py --case CASE-001 malware --vt-hash "<sha256_here>" | Tee-Object
reports/vt_hash_lookup.json
```

Salida esperada: `analysis_id`, `analysis_report` o resumen printable. `reporting.present_vt_report` puede crear un JSON `vt_result_<sha256>.json` en la raíz o en `reports/` según tu `reporting` implementado.

Explicación de los dos bloques

PowerShell (Windows)

- `python .\main.py --case CASE-001 malware --vt-upload .\test_data\notepad_sample.exe | Tee-Object reports/vt_upload_notepad.json`
  Ejecuta el script `main.py` en el contexto del caso `CASE-001`, con el subcomando `malware` y la opción `--vt-upload` sobre el archivo `.\test_data\notepad_sample.exe`. La salida del comando (respuesta del proceso de subida a VirusTotal) se imprime en consola y al mismo tiempo se guarda en el archivo `reports/vt_upload_notepad.json` mediante `Tee-Object`.
- `python .\main.py --case CASE-001 malware --vt-hash "<sha256_here>" | Tee-Object reports/vt_hash_lookup.json`
  Ejecuta el script `main.py` en el contexto del caso `CASE-001`, con el subcomando `malware` y la opción `--vt-hash`, pasando como argumento el valor SHA256 correspondiente al archivo previamente analizado. La salida se muestra en pantalla y se escribe en `reports/vt_hash_lookup.json`.

Verificación

PowerShell

1. Confirmar existencia de los archivos de salida:

```
Test-Path .\reports\vt_upload_notepad.json
Test-Path .\reports\vt_hash_lookup.json
```

2. Revisar tamaño y fecha de cada archivo:

```
Get-Item .\reports\vt_upload_notepad.json, .\reports\vt_hash_lookup.json | Select-Object
Name,Length,LastWriteTime
```

3. Ver una muestra de la salida (*primeras 20 líneas*):

```
Get-Content .\reports\vt_upload_notepad.json -TotalCount 20
Get-Content .\reports\vt_hash_lookup.json -TotalCount 20
```

Salida esperada

- En la consola y en los archivos de salida debe aparecer información relacionada con el análisis en VirusTotal.
- Se espera la presencia de campos como:
  - `analysis_id`
  - `analysis_report` o un resumen imprimible del análisis
- Dependiendo de la implementación de `reporting.present_vt_report`, puede generarse adicionalmente un archivo JSON con nombre `vt_result_<sha256>.json`, ya sea en la carpeta raíz o dentro de `reports/`.

Esto permitirá confirmar tanto la subida del archivo a VirusTotal como la consulta por hash del mismo análisis.

---

7 — FileScan.io upload (tu wrapper robusto)

Prueba el flujo de FileScan (usa tu `FILESCAN_API_KEY`); tu `filescan.py` prueba distintos headers/endpoints.

PowerShell:

```
python .\main.py --case CASE-001 malware --filescan-upload .\test_data\notepad_sample.exe | Tee-Object
reports/filescan_upload_notepad.json
```

Verifica: respuestas con `flow_id` y, si `poll_until_complete`, reporte final en `report`. Si obtienes `unauthorized` revisa qué header funcionó (tus pruebas mostraron `X-API-Key` y `x-apikey` como válidos).

Explicación de los dos bloques

PowerShell (Windows)

- `python .\main.py --case CASE-001 malware --filescan-upload .\test_data\notepad_sample.exe | Tee-Object reports/filescan_upload_notepad.json`
  Ejecuta el script `main.py` en el contexto del caso `CASE-001`, invocando el subcomando `malware` con la opción `--filescan-upload` sobre el fichero `.\test_data\notepad_sample.exe`. La salida estándar producida por la ejecución se muestra en pantalla y, al mismo tiempo, se graba en el archivo `reports/filescan_upload_notepad.json` mediante `Tee-Object`. El flujo de interacción con el servicio FileScan (incluyendo intentos con distintos headers/endpoints) es gestionado por el módulo `filescan.py` que se ejecuta como parte de `main.py`.

Verificación

Comprobar existencia del archivo de salida

```
Test-Path .\reports\filescan_upload_notepad.json
```

- Resultado esperado: `True` si la salida se guardó correctamente.

Inspeccionar tamaño y metadatos

```
Get-Item .\reports\filescan_upload_notepad.json | Select-Object Name,Length,LastWriteTime
```

Extraer y mostrar campos clave (`flow_id`, `report`) si la salida es JSON

```
# Cargar el JSON
$json = Get-Content .\reports\filescan_upload_notepad.json -Raw | ConvertFrom-Json

# Mostrar flow_id y report (si existen)
$json | Select-Object @{Name='flow_id'; Expression={$_.flow_id}}, @{Name='report'; Expression=
{$_.report}}
```

- Resultado esperado: un valor en `flow_id` y, si el flujo incluye reporte final, contenido en `report` o un objeto anidado con el reporte.

Buscar indicación de estado/terminación (polling)

```
# Si el JSON contiene un campo 'status' o 'state', mostrarlo
$json | Select-Object @{Name='status'; Expression={$_.status}}, @{Name='state'; Expression={$_.state}}
```

- Resultado esperado: campos que indiquen estado del análisis (por ejemplo `pending`, `running`, `completed`).

Comprobar texto 'unauthorized' (respuesta de autenticación)

```
Select-String -Path .\reports\filescan_upload_notepad.json -Pattern '(?i)unauthoriz|401' -SimpleMatch
```

- Resultado esperado: si aparece coincidencia, la respuesta del servicio fue de autorización rechazada.

Ver qué header (si se imprime en la salida) funcionó durante las pruebas

```
# Buscar referencias a headers habituales probadas por filescan.py
Select-String -Path .\reports\filescan_upload_notepad.json -Pattern 'X-API-Key|x-apikey|Authorization'
-AllMatches
```

- Resultado esperado: líneas donde el módulo haya registrado qué header resultó válido (por ejemplo `X-API-Key` o `x-apikey`). Si no hay coincidencias, el módulo no volcó explícitamente esa información en la salida.

Mostrar una muestra legible del JSON (primeras 40 líneas)

```
Get-Content .\reports\filescan_upload_notepad.json -TotalCount 40
```

Salida esperada (en la verificación)

- Presencia de `flow_id` en la respuesta (identificador del flujo iniciado).
- Si `poll_until_complete` se ejecutó y el servicio terminó, existencia de un `report` o un objeto final con el resultado del análisis.
- Si ocurrió un error de autenticación, aparición de `unauthorized`, `401` o similar en la salida.
- Si el módulo dejó rastro del header efectivo, aparición de `X-API-Key` o `x-apikey` en la salida.

---

8 — Módulo *osint* (subdomains / ip-info / email-leaks)

Ejemplos (depende de lo implementado en `modules/osint.py` — si son stubs, verás mensajes informativos):

PowerShell:

```
python .\main.py osint --subdomains example.com | Tee-Object reports/osint_subdomains_example.json
```

```
python .\main.py osint --ip-info 8.8.8.8 | Tee-Object reports/osint_ip_8.8.8.8.json

python .\main.py osint --email-leaks me@example.com | Tee-Object reports/osint_email_me_example.json
```

Explicación de los dos bloques

PowerShell (Windows)

- `python .\main.py osint --subdomains example.com | Tee-Object reports/osint_subdomains_example.json`
  Ejecuta `main.py` con el subcomando `osint`, solicitando la enumeración de subdominios para `example.com`. La salida del proceso se muestra en pantalla y se guarda simultáneamente en `reports/osint_subdomains_example.json` mediante `Tee-Object`.
- `python .\main.py osint --ip-info 8.8.8.8 | Tee-Object reports/osint_ip_8.8.8.8.json`
  Ejecuta `main.py` con el subcomando `osint`, solicitando información OSINT asociada a la dirección IP `8.8.8.8`. La salida se duplica en pantalla y en el archivo `reports/osint_ip_8.8.8.8.json`.
- `python .\main.py osint --email-leaks me@example.com | Tee-Object reports/osint_email_me_example.json`
  Ejecuta `main.py` con el subcomando `osint`, consultando posibles filtraciones o leaks para el correo `me@example.com`. La salida se muestra en consola y se graba en `reports/osint_email_me_example.json`.

Verificación

1. Comprobar existencia de los archivos de salida:

```
Test-Path .\reports\osint_subdomains_example.json
Test-Path .\reports\osint_ip_8.8.8.8.json
Test-Path .\reports\osint_email_me_example.json
```

- Resultado esperado: `True` si cada comando guardó su salida.

2. Ver tamaño y metadatos de los archivos:

```
Get-Item .\reports\osint_subdomains_example.json, .\reports\osint_ip_8.8.8.8.json,
.\reports\osint_email_me_example.json |
  Select-Object Name,Length,LastWriteTime
```

- `Length` > 0 indica que hay contenido escrito.

3. Mostrar una **muestra** (primeras 25 líneas) de cada archivo:

```
Get-Content .\reports\osint_subdomains_example.json -TotalCount 25
Get-Content .\reports\osint_ip_8.8.8.8.json -TotalCount 25
Get-Content .\reports\osint_email_me_example.json -TotalCount 25
```

- Resultado esperado: líneas con datos legibles (objetos JSON, listas de subdominios, bloques con información de IP o cadenas indicando leaks) o un mensaje indicando que la funcionalidad está *stubbed*.

4. Buscar indicación explícita de que la funcionalidad está stubbed (palabras clave comunes):

```
Select-String -Path .\reports\osint_*.json -Pattern 'stub|not implemented|stubbed|placeholder|no data'
-SimpleMatch
```

- Resultado esperado: coincidencias si alguna salida contiene un aviso de funcionalidad no implementada.

5. Extraer y mostrar campos JSON relevantes (si el archivo está en formato JSON):

```
# Cargar JSON y mostrar keys principales para cada archivo
$json1 = Get-Content .\reports\osint_subdomains_example.json -Raw | ConvertFrom-Json -ErrorAction
SilentlyContinue
$json2 = Get-Content .\reports\osint_ip_8.8.8.8.json -Raw | ConvertFrom-Json -ErrorAction
SilentlyContinue
$json3 = Get-Content .\reports\osint_email_me_example.json -Raw | ConvertFrom-Json -ErrorAction
```

```
SilentlyContinue

$json1 | Get-Member -MemberType NoteProperty | Select-Object Name
$json2 | Get-Member -MemberType NoteProperty | Select-Object Name
$json3 | Get-Member -MemberType NoteProperty | Select-Object Name
```

- Resultado esperado: nombres de campos como `subdomains`, `ip_info`, `leaks`, `message`, etc., o ningún resultado si el archivo no era JSON válido.

6. (Opcional) Buscar evidencia de uso de Shodan/u otras fuentes en la salida:

```
Select-String -Path .\reports\osint_*.json -Pattern 'shodan|shodan.io|obase' -CaseSensitive:$false
```

- Resultado esperado: líneas donde se mencione la fuente (`shodan`, `obase`) si el módulo registró su origen de datos.

---

9 — Módulo *guides* (listar / mostrar guías)

PowerShell:

```
python .\main.py guides --list

python .\main.py guides --show quickstart | Tee-Object reports/guides_quickstart.txt
```

Verifica: que muestre listas y contenido (si `guides` contiene archivos).

---

10 — Módulos *attack* / *defend* (pruebas no destructivas)

Pruebas de ejemplo (sin acciones destructivas):

PowerShell:

```
python .\main.py attack --scan-ports 127.0.0.1
python .\main.py defend --check-firewall
```

Bash:

```
python3 main.py attack --scan-ports 127.0.0.1
python3 main.py defend --check-processes
```

Explicación de los dos bloques

PowerShell (Windows)

- `python .\main.py attack --scan-ports 127.0.0.1`
  Invoca el script `main.py` con el subcomando `attack` y la opción `--scan-ports` apuntando a la dirección `127.0.0.1`. El comando solicita realizar (o simular) un escaneo de puertos sobre `localhost` y muestra la salida resultante en la consola.
- `python .\main.py defend --check-firewall`
  Ejecuta `main.py` con el subcomando `defend` y la opción `--check-firewall`. El comando solicita comprobar el estado/configuración del firewall en la máquina local y presenta en la consola la información resultante.

Bash (Linux / macOS)

- `python3 main.py attack --scan-ports 127.0.0.1`
  Ejecuta `main.py` usando `python3` con el subcomando `attack` y la opción `--scan-ports` sobre `127.0.0.1`. Se espera que el proceso realice (o simule) un escaneo de puertos en `localhost` y devuelva la salida por stdout.

- `python3 main.py defend --check-processes`
  Invoca `main.py` con el subcomando `defend` y la opción `--check-processes`. Se solicita listar o verificar procesos relevantes al componente de defensa en la máquina local; la información se imprime en la consola.

Qué confirmar visualmente

- Para `--scan-ports`: presencia de una lista de puertos con su estado (open/closed/filtered) o texto que indique que es una simulación/placeholder.
- Para `--check-firewall` / `--check-processes`: salida con estado del firewall o listado de procesos; si la funcionalidad está stubbed, aparecerá un mensaje explícito (por ejemplo `not implemented`, `stubbed`, `placeholder`) o una estructura mínima informativa. **salidas informativas o listas (es normal que sean placeholders).**

---

11 — Guardado automático y ubicación de reportes

- Todos los `Tee-Object` / `tee` del ejercicio guardan salidas en `reports/`.
- Si `modules/reporting.py` implementa `save_json_report` o `present_vt_report`, asegurarse que escriba en `reports/` con nombre:

```
reports/_.json
```

Si no existe, los comandos anteriores guardan stdout en `reports/` y sirven como evidencia.

---

12 — Scripts automáticos (ejecutan todo)

PowerShell: `scripts/run_all_tests.ps1`

Crear el archivo con este contenido (pegar en `scripts/run_all_tests.ps1`):

```
param(
  [string]$Case = "CASE-001"
)

$base = (Get-Location).Path
$td = Join-Path $base "test_data"
$p_notepad = (Resolve-Path (Join-Path $td "notepad_sample.exe")).Path
$p_random = (Resolve-Path (Join-Path $td "random.bin")).Path

New-Item -ItemType Directory -Path (Join-Path $base "reports") -Force | Out-Null

Write-Host "1) Forensics"
python .\main.py forensics --file-hash $p_notepad | Tee-Object -FilePath reports/forensics_notepad.json

Write-Host "2) Malware strings / entropy"
```

```
python .\main.py malware --strings $p_notepad | Tee-Object -FilePath
reports/malware_strings_notepad.txt

python .\main.py malware --entropy $p_random | Tee-Object -FilePath reports/malware_entropy_random.txt



Write-Host "3) Malware tool (objdump)"

python .\main.py malware --tool objdump --target $p_notepad | Tee-Object -FilePath
reports/malware_tool_objdump.txt



Write-Host "4) Deep scan battery"

python .\main.py malware --deep-scan $p_notepad | Tee-Object -FilePath
reports/malware_deepscan_notepad.json



Write-Host "5) VT upload (opcional)"

python .\main.py --case $Case malware --vt-upload $p_notepad | Tee-Object -FilePath
reports/vt_upload_notepad.json



Write-Host "6) FileScan upload (opcional)"

python .\main.py --case $Case malware --filescan-upload $p_notepad | Tee-Object -FilePath
reports/filescan_upload_notepad.json



Write-Host "7) OSINT & Guides & Attack/Defend quick checks"

python .\main.py osint --ip-info 8.8.8.8 | Tee-Object -FilePath reports/osint_ip_8.8.8.8.json

python .\main.py guides --list | Tee-Object -FilePath reports/guides_list.txt

python .\main.py attack --scan-ports 127.0.0.1 | Tee-Object -FilePath reports/attack_scan_ports.txt

python .\main.py defend --check-firewall | Tee-Object -FilePath reports/defend_check_firewall.txt



Write-Host "All done. Check reports/"
```

Ejecutar:

```
powershell -ExecutionPolicy Bypass -File .\scripts\run_all_tests.ps1
```

Bash: `scripts/run_all_tests.sh`

```
#!/usr/bin/env bash

set -euo pipefail

mkdir -p reports



p_notepad="$(realpath test_data/notepad_sample.exe)"

p_random="$(realpath test_data/random.bin)"
```

```
echo "1) Forensics"
python3 main.py forensics --file-hash "$p_notepad" | tee reports/forensics_notepad.json

echo "2) Malware strings / entropy"
python3 main.py malware --strings "$p_notepad" | tee reports/malware_strings_notepad.txt
python3 main.py malware --entropy "$p_random" | tee reports/malware_entropy_random.txt

echo "3) Malware tool (objdump)"
python3 main.py malware --tool objdump --target "$p_notepad" | tee reports/malware_tool_objdump.txt

echo "4) Deep scan battery"
python3 main.py malware --deep-scan "$p_notepad" | tee reports/malware_deepscan_notepad.json

echo "5) VT upload (optional)"
python3 main.py --case CASE-001 malware --vt-upload "$p_notepad" | tee reports/vt_upload_notepad.json
|| true

echo "6) FileScan upload (optional)"
python3 main.py --case CASE-001 malware --filescan-upload "$p_notepad" | tee
reports/filescan_upload_notepad.json || true

echo "7) OSINT & Guides & Attack/Defend quick checks"
python3 main.py osint --ip-info 8.8.8.8 | tee reports/osint_ip_8.8.8.8.json || true
python3 main.py guides --list | tee reports/guides_list.txt || true
python3 main.py attack --scan-ports 127.0.0.1 | tee reports/attack_scan_ports.txt || true
python3 main.py defend --check-firewall | tee reports/defend_check_firewall.txt || true

echo "Done. Check reports/"
```

Hacerlo ejecutable y ejecutar:

```
chmod +x scripts/run_all_tests.sh
./scripts/run_all_tests.sh
```

Explicación de los dos bloques

PowerShell: `scripts/run_all_tests.ps1`
Define un parámetro `$Case`, resuelve rutas a `test_data/notepad_sample.exe` y `test_data/random.bin`, crea `reports/` y ejecuta secuencialmente una batería de pruebas numeradas: forensics, extracción de *strings* y cálculo de entropía, ejecución de `objdump`, escaneo profundo (`--deep-scan`), subidas opcionales a VT y FileScan (en contexto de caso), y comprobaciones OSINT/guides/attack/defend. Cada paso duplica la salida en consola y la guarda en `reports/...`. Al final imprime `All done. Check reports/`.

Comando de ejecución:
`powershell -ExecutionPolicy Bypass -File .\scripts\run_all_tests.ps1`

Bash: `scripts/run_all_tests.sh`
Crea `reports/`, resuelve rutas absolutas a `test_data/notepad_sample.exe` y `test_data/random.bin`, y ejecuta la misma batería de pruebas que el script PowerShell (con `python3`). Usa `set -euo pipefail` y marca las operaciones opcionales con `|| true` para que no detengan todo si fallan. Al final imprime `Done. Check reports/`.

Verificación

- Comprobar que existe la carpeta `reports` y listarla:
  PowerShell: `Test-Path .\reports` / `Get-ChildItem .\reports`
  Bash: `[ -d reports ] && ls -l reports`
- Verificar archivos clave creados (ejemplos):
  PowerShell:

```
Test-Path .\reports\forensics_notepad.json
Get-Item .\reports\forensics_notepad.json | Select Name,Length,LastWriteTime
Get-Content .\reports\forensics_notepad.json -TotalCount 10
```

Bash:

```
[ -f reports/forensics_notepad.json ] && ls -lh reports/forensics_notepad.json head -n 10
reports/forensics_notepad.json
```

---

13 — Interpretación rápida de resultados (qué buscar)

- `forensics_notepad.json` → toma el `sha256` que usarás para VT lookup.
- `malware_strings_notepad.txt` → confirma que el extractor Python funciona (muestra strings).
- `malware_entropy_random.txt` → entropía alta para `random.bin`.
- `malware_tool_objdump.txt` → si `objdump` está instalado mostrará disassembly; si no, verás `executable not found`.
- `malware_deepscan_notepad.json` → colección `analyses` con cada herramienta: rc/stdout/stderr o `executable not found`.
- `vt_upload_notepad.json` / `vt_hash_lookup.json` → `analysis_id`, `analysis_report`, `last_analysis_stats`.
- `filescan_upload_notepad.json` → `flow_id` y `report` (dependiendo de la API).
- `reports/*` → sirven como evidencia reproducible.

---

14 — Solución de Problemas Comunes

- **El placeholder aparece 0 bytes**: OneDrive/AV puede bloquear. Crea el archivo con `Out-File`/`printf` como en la sección 1 y verifica permisos. Si OneDrive sincroniza, excluye la carpeta o usa una carpeta local fuera de OneDrive.
- **Commands** `executable not found`: instala la herramienta o ignóralo — es parte de la prueba (asegura robustez).
- **VT/FileScan 401**: revisa qué header funciona (tus pruebas mostraron `X-API-Key` o `x-apikey`). Ajusta `modules/filescan.py` si es necesario (ya lo hiciste).
- **PowerShell interpreta texto con** `[` **o emojis**: no pegues la salida de consola como comando; ejecuta los scripts.