# FIRMA DIGITAL CON EL DNIE

Iván Ciudad y Víctor Carbajo

# Conectividad con el DNIe

```python
# Importamos la librería principal para la comunicación PKCS#11
import pkcs11
import getpass
# Importamos algunos tipos de objetos y atributos que nos ayudarán a buscar cosas
from pkcs11 import ObjectClass, Attribute

# --- CONFIGURACIÓN ---
# Ruta a la librería PKCS#11 (OpenSC en Windows)
LIB_PATH = 'C:/Archivos de Programa/OpenSC Project/OpenSC/pkcs11/opensc-pkcs11.dll'

# --- CÓDIGO PRINCIPAL ---
try:
    # Cargamos la librería PKCS#11 desde la ruta que hemos definido
    lib = pkcs11.lib(LIB_PATH)

    # Obtenemos la primera "ranura" (slot) donde hay un token (el DNIe) presente
    slots = lib.get_slots(token_present=True)
    if not slots:
        raise Exception("No se detecta ningún token en el lector.")

    slot = slots[0]
    token = slot.get_token()

    print("✅ DNIe detectado correctamente.")
    print(f"   - Slot: {slot}")
    print(f"   - Token (DNIe): {token}")

    # Abrimos una sesión con el token
    pin = getpass.getpass("Introduce el PIN de tu DNIe: ")
    with token.open(user_pin=pin) as session:
        print("\n✅ Sesión iniciada con éxito.")

except Exception as e:
    print(f"❌ Error: {e}")
    print("   Asegúrate de que el DNIe está insertado, la ruta de la librería es correcta y el PIN es válido.")
```

La función `get_session` (presente en todos los scripts de operación) establece la comunicación con el DNIe a través del estándar PKCS#11.

Antes de solicitar el PIN, el código realiza dos pasos esenciales para verificar el entorno:

1. **Carga de la Librería:** Utiliza `pkcs11.lib(LIB_PATH)` para cargar la librería OpenSC. `LIB_PATH` es una ruta de instalación fija (*hardcoded*) que debe ajustarse al sistema operativo del usuario. OpenSC actúa como el *middleware* o traductor de bajo nivel entre Python y el hardware.

2. **Detección del DNIe:** Llama a `lib.get_slots(token_present=True)` para encontrar si hay una ranura (*slot*) ocupada por un *token* (el DNIe). Si no se detecta ningún *token*, lanza una excepción.

Solo después de estos pasos, el script pide el PIN para abrir una sesión (`token.open(user_pin=pin)`)

# Operación de Firma

```python
def sign(archivos, pin):
    """Firma uno o varios archivos con la clave privada del DNIe."""
    if not archivos:
        print("⚠️ No se ha especificado ningún archivo para firmar.")
        print("   Uso: python dnie_cli.py sign <archivo1> <archivo2> ...")
        return
    try:
        with get_session(pin) as session:
            # --- PASO 1: Encontrar el certificado de firma primero ---
            print("🔍 Buscando el certificado de firma digital...")
            cert_firma = None
            for cert in session.get_objects({Attribute.CLASS: ObjectClass.CERTIFICATE}):
                label = cert[Attribute.LABEL]
                if label and "firmadigital" in label.replace(" ", "").lower():
                    cert_firma = cert
                    break

            if not cert_firma:
                raise Exception("No se encontró el certificado de firma digital.")

            # --- PASO 2: Obtener el ID único del certificado ---
            cert_id = cert_firma[Attribute.ID]
            print(f"✅ Certificado encontrado. Buscando clave privada asociada (ID: {bytes(cert_id).hex()})...")
            priv_key = next(session.get_objects({
                Attribute.CLASS: ObjectClass.PRIVATE_KEY,
                Attribute.ID: cert_id  # La clave es buscar por el ID que hemos obtenido
            }))

            print("✅ Clave privada encontrada. Procediendo a firmar archivos...")

            # --- PASO 3: Bucle para firmar cada archivo ---
            for archivo in archivos:
                try:
                    output_path = f"{archivo}.sig"

                    print(f"\n📄 Firmando '{archivo}'...")

                    with open(archivo, 'rb') as f:
                        data = f.read()

                    signature = priv_key.sign(data, mechanism=Mechanism.SHA256_RSA_PKCS)

                    with open(output_path, 'wb') as f:
                        f.write(signature)

                    print(f"✅ Archivo '{archivo}' firmado correctamente.")
                    print(f"   Firma guardada en '{output_path}'")

                except Exception as e:
                    # Si un archivo falla, informa del error y continúa con el siguiente
                    print(f"   ❌ Error al firmar el archivo '{archivo}': {e}")
```

**1. Establecimiento de la Sesión y PIN** Se inicia la sesión con el DNIe (`token.open()`) después de que el usuario introduce y se valida el PIN.

**2. Búsqueda Segura de la Clave Privada** Para asegurarse de que se utiliza la clave de *firma* (y no, por ejemplo, la clave de autenticación), el código sigue un proceso de dos pasos vinculando el certificado a su clave privada:

• **Identificación del Certificado:** Se buscan todos los objetos de clase `CERTIFICATE` almacenados en la tarjeta. Se itera sobre ellos para encontrar el que tiene la etiqueta (`Attribute.LABEL`) que corresponde al certificado de firma, que usualmente es "CertFirmaDigital".

• **Vinculación por ID:** Una vez encontrado el certificado de firma (`cert_firma`), se extrae su identificador único (`Attribute.ID`). Luego, se realiza una nueva búsqueda para localizar un objeto de clase `PRIVATE_KEY` que posea **exactamente el mismo** `Attribute.ID`. Esto garantiza la selección de la clave correcta.

**3. Ejecución de la Criptografía (Dentro del Chip)** Una vez encontrada la clave privada (`priv_key`):

• Se leen los *bytes* (`data`) del archivo a firmar.

• Se invoca la operación `priv_key.sign(data, mechanism=Mechanism.SHA256_RSA_PKCS)`.

• **Seguridad:** El punto fundamental es que la clave privada **nunca abandona el chip del DNIe**. La aplicación solo envía los datos al DNIe, y el hardware de la tarjeta realiza internamente el cálculo del hash SHA-256 y su cifrado con la clave privada RSA.

**4. Resultado** La aplicación recibe el resultado de la operación (la firma) y la guarda como una firma "detached" (separada) en un archivo con extensión `.sig`

# Exportación de certificado

```python
def export_cert(pin, output):
    """Exporta el certificado de firma digital del DNIe."""
    try:
        with get_session(pin) as session:
            print("🔍 Buscando certificados en el DNIe...")

            certificados = list(session.get_objects({Attribute.CLASS: ObjectClass.CERTIFICATE}))
            if not certificados:
                raise Exception("No se encontraron certificados en el DNIe.")

            cert_firma = None
            for cert in certificados:
                label = cert[Attribute.LABEL]
                print(f"   - {label}")
                if label and "firmadigital" in label.replace(" ", "").lower():
                    cert_firma = cert
                    break

            if not cert_firma:
                raise Exception("No se encontró el certificado de firma digital (CertFirmaDigital).")

            cert_data = cert_firma[Attribute.VALUE]
            with open(output, 'wb') as f:
                f.write(cert_data)

            print(f"✅ Certificado exportado correctamente: {output}")

    except Exception as e:
        print(f"❌ Error al exportar certificado: {e}")
```

El proceso de exportación ocurre inmediatamente después de establecer la sesión con el DNIe y validar el PIN.

1. **Inicio de la Sesión:** Se utiliza la función `get_session(pin)` para cargar la librería PKCS#11, detectar el DNIe y abrir una sesión.

2. **Búsqueda de Objetos:** Una vez abierta la sesión, el código busca todos los objetos que pertenecen a la clase `ObjectClass.CERTIFICATE` dentro de la tarjeta (`list(session.get_objects({Attribute.CLASS: ObjectClass.CERTIFICATE}))`).

3. **Identificación del Certificado de Firma:**

   ◦ Se itera sobre los certificados encontrados.

   ◦ El código identifica el certificado de firma buscando aquel cuya etiqueta (`Attribute.LABEL`) contenga la cadena "firmadigital" (ignorando espacios y mayúsculas, como se ve en el código `dnie_cli.py`). Esto se hace para encontrar la etiqueta asignada por el DNIe, que suele ser "CertFirmaDigital".

4. **Extracción del Contenido:** Una vez localizado el certificado correcto (`cert_firma`), se extrae su contenido binario utilizando el atributo `Attribute.VALUE`. Este contenido es el certificado en formato estándar DER.

5. **Guardado:** Los datos binarios (`cert_data`) se escriben en un archivo de salida con extensión `.der`.

# Verificar la firma

```python
def verify(documento, firma, certificado):
    """Verifica la firma de un DOCUMENTO usando la FIRMA y el CERTIFICADO (DER)."""
    try:
        # --- Leer archivos ---
        with open(documento, 'rb') as f:
            data = f.read()
        with open(firma, 'rb') as f:
            signature = f.read()
        with open(certificado, 'rb') as f:
            cert_bytes = f.read()

        # --- Cargar certificado ---
        cert = x509.load_der_x509_certificate(cert_bytes)

        print("\n=== INFORMACIÓN DEL CERTIFICADO ===")
        print(f"📄 Sujeto: {cert.subject}")
        print(f"🏢 Emisor: {cert.issuer}")
        print(f"📅 Válido desde: {cert.not_valid_before_utc.strftime('%Y-%m-%d %H:%M:%S')} UTC")
        print(f"📅 Válido hasta: {cert.not_valid_after_utc.strftime('%Y-%m-%d %H:%M:%S')} UTC")

        # --- Calcular hash para mostrarlo ---
        hash_documento = hashlib.sha256(data).hexdigest()
        print(f"\n🔐 Hash del documento (SHA-256): {hash_documento}")

        # --- Intentar verificación con cryptography (MÉTODO SIMPLIFICADO Y ROBUSTO) ---
        print("\n🔍 Intentando verificación con librería 'cryptography'...")
        try:
            public_key = cert.public_key()
            # Dejamos que la librería haga el hash internamente. Es más fiable.
            public_key.verify(
                signature,
                data,  # Pasamos los datos originales, no el hash
                padding.PKCS1v15(),
                hashes.SHA256() # Indicamos el algoritmo de hash que debe usar
            )
            print("\n✅✅✅ VERIFICACIÓN EXITOSA ✅✅✅")
            return
        except InvalidSignature:
            print("⚠️  Verificación con 'cryptography' falló. La firma no es válida.")
        except Exception as e:
            print(f"⚠️  Ocurrió un error inesperado con 'cryptography': {e}")

    except Exception as e:
        print(f"\n❌❌❌ ERROR GENERAL ❌❌❌")
        print(f"No se pudo completar la verificación: {e}")
```

**1. Requisitos de Entrada** El script `verificar_firma_dnie.py` requiere tres archivos de entrada obligatorios:

• El **documento original** (`documento`).

• El archivo de **firma** binaria separada o "detached" (`.sig`).

• El **certificado público** del firmante (`.der`), obtenido previamente mediante la operación de exportación.

**2. Carga y Extracción del Certificado**

• Se lee el contenido binario del certificado (`cert_bytes`).

• Se utiliza `x509.load_der_x509_certificate(cert_bytes)` para cargar el certificado público.

• Se extrae la clave pública del certificado (`public_key = cert.public_key()`).

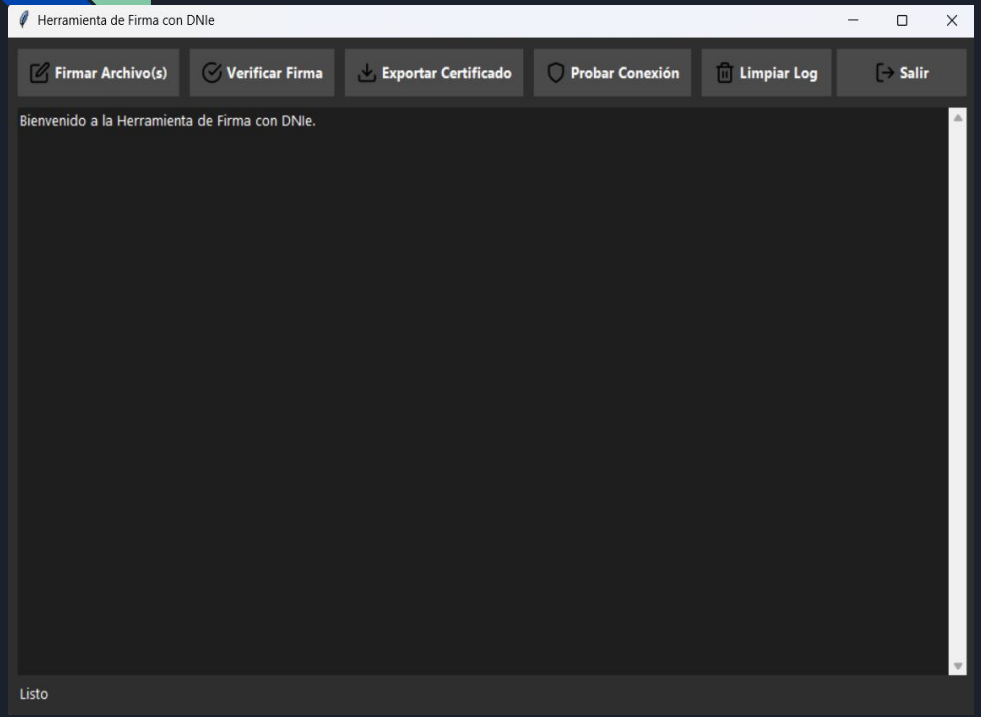
**3. La Verificación Criptográfica (Función `verify()`)** . La clave pública se utiliza para invocar el método `verify()`:

• **Descifrado del Hash:** La función descifra la firma digital (`signature`) usando la clave pública, lo que revela el hash SHA-256 original que fue cifrado por el DNIe.

• **Recálculo del Hash:** La función calcula un *nuevo* hash SHA-256 del documento original (`data`) proporcionado.

• **Comparación:** Finalmente, compara ambos hashes. Si son idénticos, significa que la firma es auténtica (generada por la clave privada asociada al certificado) y el documento no ha sido modificado.

# Interfaz de usuario




1. **Threading (Multihilo) para las Operaciones Críticas:**

   ◦ Cualquier operación que interactúa con el DNIe (como firmar, exportar o el diagnóstico) se ejecuta en un hilo separado (`threading.Thread`).

   ◦ **¿Por qué es clave?** Las operaciones con el DNIe requieren esperar la respuesta del hardware. Al ejecutarlas en un hilo separado, se evita que el hilo principal de la GUI (`tkinter`) se bloquee, garantizando que la ventana permanezca activa y receptiva.

2. **Manejo Seguro de la GUI (`root.after(0, ...)`)**:

   ◦ Para evitar errores de concurrencia al actualizar elementos visuales (como la etiqueta de estado o el área de *log*) desde los hilos secundarios, el código usa `self.root.after(0, self._log_message, message)`.

   ◦ Esto pone la tarea de actualización en una cola para que sea ejecutada por el hilo principal de `tkinter` cuando sea seguro.

3. **Diseño de Experiencia de Usuario (UX) Avanzada:**

   ◦ La versión `intercomp_v1.py` se enfoca en una UX moderna con un tema oscuro y utiliza librerías como `cairosvg` y `Pillow` para cargar iconos SVG. Esto, sin embargo, introduce fuertes dependencias externas

# Limitaciones:

**Ruta de Librería Fija**

**Formato de Firma Básico**

**Alcance Limitado de Verificación**

# Firmando con el DNIe