Gestor de Contraseñas con DNIe

Almacenamiento Seguro de Credenciales con Autenticación 2FA

Daniel Modrego & Ana Lushan Montuenga

Seguridad en Redes y Servicios

Contexto del Proyecto

Características

- Autenticación 2FA con DNIe
- Almacenamiento encriptado de credenciales
- Uso de técnicas criptográficas modernas
- Soporte multi-usuario
- Generador aleatorio de contraseñas
- Interfaz gráfica

Tecnologías y Librerías

- Lenguaje: Python 3.x
- Criptografía: cryptography, argon2-cffi
- Smart card: python-pkcs11, OpenSC
- Gestión memoria: zeroize, mlock
- Interfaz: PyQt5

Propiedades de Seguridad del Sistema

Confidencialidad

Garantizada mediante:

- Fernet (AES-128-CBC)
- Base de datos encriptada en disco
- Memory locking

Autenticidad

Garantizada mediante:

- 2FA obligatorio
- RSA-PKCS#1 (DNIe)
- Argon2id (contraseña)

Integridad

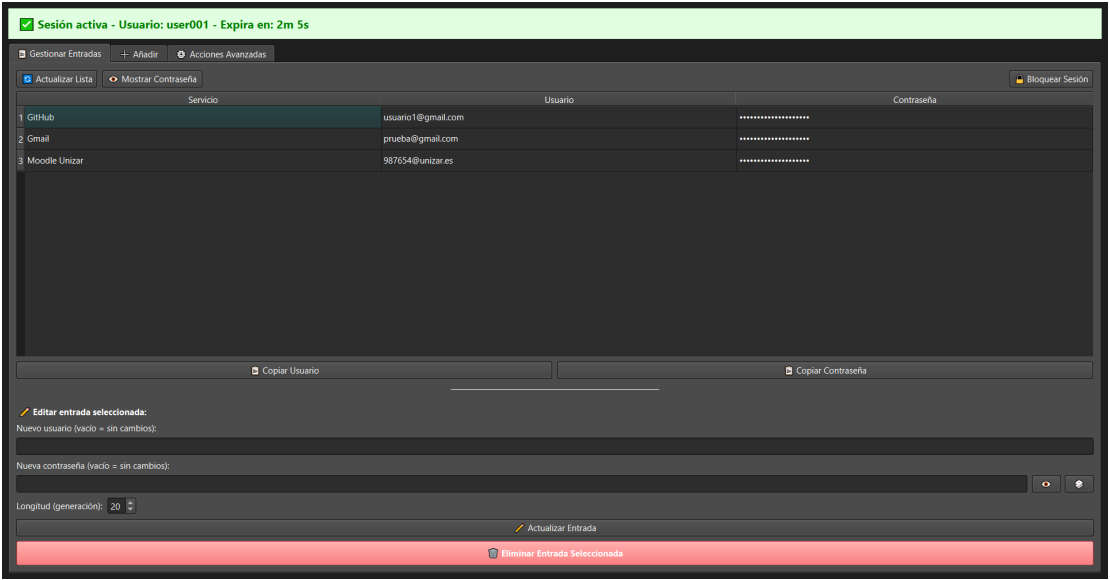
Garantizada mediante:

- HMAC-SHA256
- Verificación al desencriptar (Fernet)

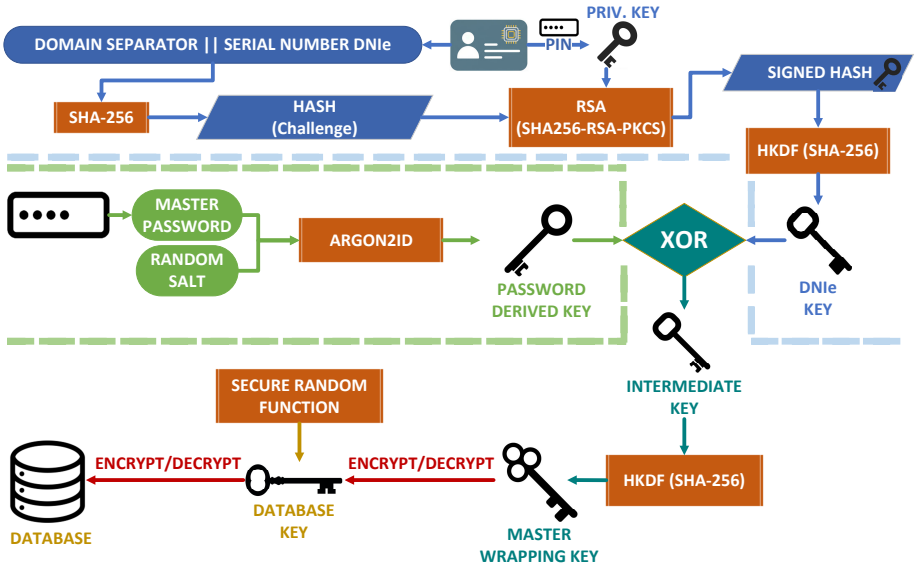
Propiedad Adicional

Forward Secrecy mediante rotación automática de K_{db} al cerrar sesión

DEMO


Sesión activa - Usuario: user001 - Expira en: 2m 5s
Gestionar Entradas
Añadir
Acciones Avanzadas
Actualizar Lista
Mostrar Contraseña
Bloquear Sesión
Servicio
Usuario
Contraseña
1 GitHub
usuario1@gmail.com
2 Gmail
prueba@gmail.com
3 Moodle Unizar
987654@unizar.es
Copiar Usuario
Copiar Contraseña
Editar entrada seleccionada:
Nuevo usuario (vacío = sin cambios):
Nueva contraseña (vacío = sin cambios):
Longitud (generación):
20
Actualizar Entrada
Eliminar Entrada Seleccionada

Sistema de autenticación y cifrado



Autenticación de Dos Factores

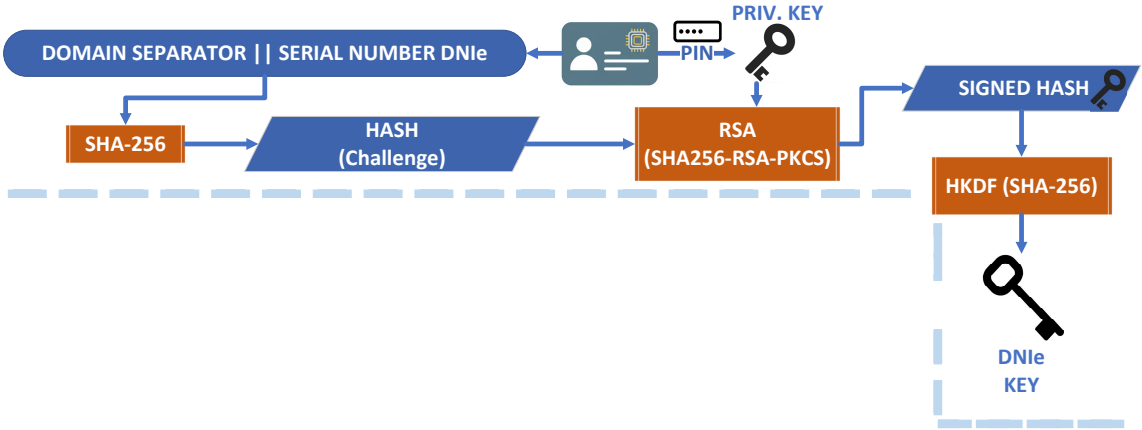
Factor 1: DNIe (Posesión)

- Firma digital con clave privada RSA del DNIe
- Challenge determinístico basado en SHA-256
- Derivación de clave mediante HKDF

Factor 2: Contraseña Maestra (Conocimiento)

- Derivación con Argon2id (memory-hard)
- 64 MiB de memoria, 3 iteraciones
- Protección contra ataques de fuerza bruta

Firma Digital con DNIe



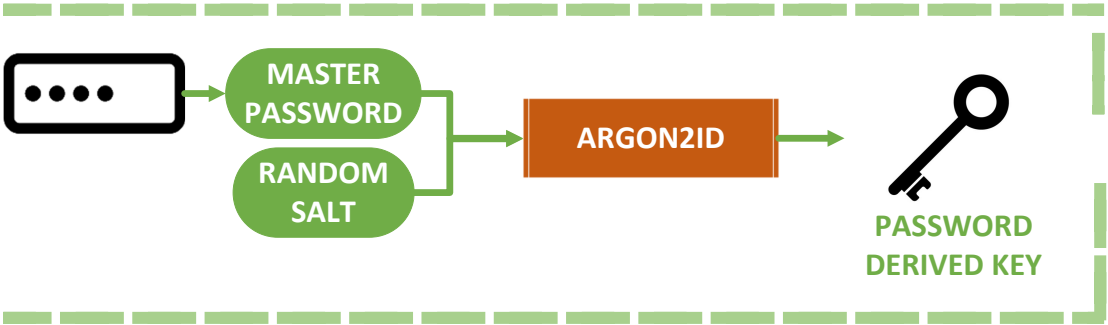
```
# Challenge deterministico (SHA-256)
digest = hashes.Hash(hashes.SHA256(), backend=default_backend())
digest.update(b'password-manager-signature-challenge-v1:')
digest.update(card_serial.encode('utf-8'))
challenge = bytearray(digest.finalize())

# Firma con clave privada del DNIe (RSA-PKCS#1)
signature = bytearray(private_key.sign(
        bytes(challenge),
        mechanism=Mechanism.SHA256_RSA_PKCS
))

# Derivacion de clave de wrapping con HKDF
kdf = HKDF(algorithm=hashes.SHA256(), length=32,
        salt=b'dnie-signature-wrapping-key-v1',
        info=b'database-key-protection')
wrapping_key = kdf.derive(bytes(signature))
```

Listing 1: Challenge y firma RSA

Derivación de Clave con Argon2id

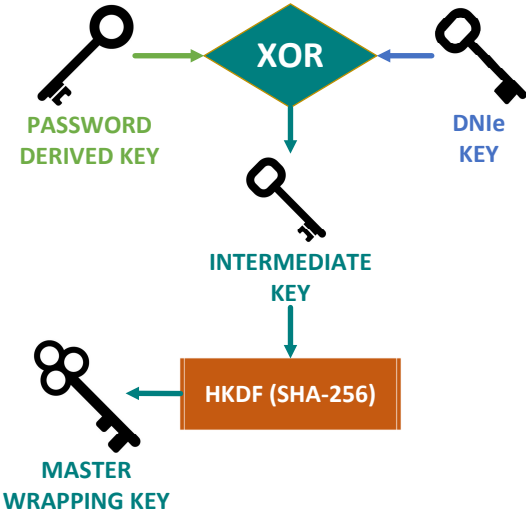


Derivación de Clave con Argon2id

```python
def derive_key_from_password(password: str, salt: bytes) -> bytes:
        key = hash_secret_raw(
                secret=password.encode("utf-8"),
                salt=salt,
                time_cost=3,            # 3 iteraciones
                memory_cost=65536,      # 64 MiB
                parallelism=2,          # 2 hilos
                hash_len=32,            # 32 bytes
                type=Type.ID,           # Argon2id
        )
        return key
```

Listing 2: Derivación desde contraseña maestra

Combinación de Claves (XOR + HKDF)

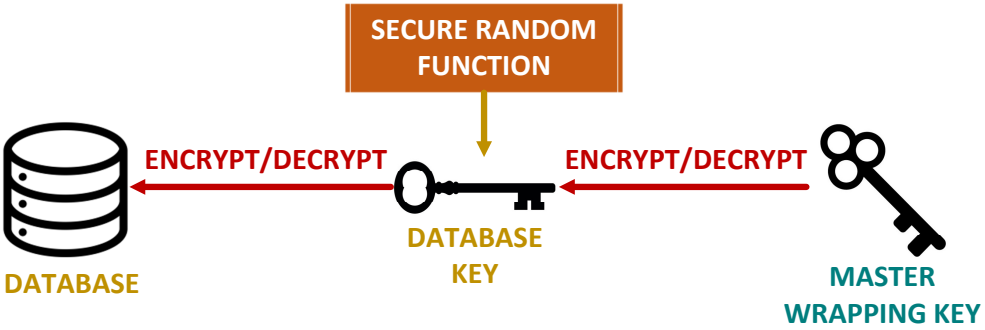


Combinación de Claves (XOR + HKDF)

```
def combine_keys(dnie_key, password_key):
        # XOR de las dos claves
        xor_key = bytearray(a ^ b for a, b in zip(dnie_key, password_key))
        try:
                # HKDF para distribucion uniforme
                kdf = HKDF(
                        algorithm=hashes.SHA256(),
                        length=32,
                        salt=b'dnie-password-combination',
                        info=b'two-factor-database-key'
                )
                combined_key = kdf.derive(bytes(xor_key))
                return combined_key
        finally:
                zeroize1(xor_key)  # Limpieza segura
                del xor_key
```

Listing 3: Fusión de factores de autenticación

Proceso de Encriptación



Proceso de Encriptación

1. **Generación**: K_{db} aleatoria (32 bytes)
2. **Wrapping**:

$$K_{master} = \text{HKDF}(K_{DNIe} \oplus K_{password})$$

$$K_{db\_wrapped} = \text{Fernet}(K_{db}, K_{master})$$

3. **Encriptación de datos**:

$$\text{Datos\_cifrados} = \text{Fernet}(K_{db}, \text{JSON})$$

Wrapping de Clave de Base de Datos

```
def wrap_database_key(k_db, dnie_key, password_key):
        # Combinar claves de autenticacion
        master_key = bytearray(combine_keys(dnie_key, password_key))

        try:
                fernet_key = bytearray(base64.urlsafe_b64encode(bytes(
                    master_key)))
                try:
                        f = Fernet(bytes(fernet_key))
                        # Encriptar K_db con Fernet
                        wrapped = f.encrypt(k_db)
                        return wrapped
                finally:
                        zeroize1(fernet_key)
                        del fernet_key, f
        finally:
                zeroize1(master_key)
                del master_key
```

Listing 4: Protección de K_{db} con Fernet

Resumen de Técnicas Criptográficas

Derivación de Claves

- Argon2id
- HKDF-SHA256
- RSA-PKCS#1

Encriptación

- Fernet (AES-128-CBC)

Autenticación

- Fernet (HMAC-SHA256)

Rotación Automática de Claves

1. **Pre-generación**: K_{db_nueva} al iniciar sesión
2. **Wrapping anticipado**: K_{db_nueva} guardada en disco
3. **Cierre de sesión**:
   - Desencriptar DB con K_{db_actual}
   - Re-encriptar DB con K_{db_nueva}
   - Reemplazar archivo wrapped key
4. **Resultado**: Forward secrecy

Rotación Automática de Claves

```python
def auto_rotate_on_logout(session):
        # 1. Desencriptar DB con clave actual
        current_db = EncryptedDatabase(bytes(session.fernet_key), db_file)
        db_content = current_db._decrypt_db()
        del current_db

        # 2. Re-encriptar DB con clave nueva (pre-generada)
        new_db = EncryptedDatabase(bytes(session.fernet_key_next), db_file)
        new_db.encrypted_data = new_db._encrypt_db(db_content)
        new_db._save_encrypted()
        del new_db

        # 3. Actualizar archivo wrapped_key
        os.replace(temp_wrapped_key, wrapped_key_file)

        # 4. Zeroizar clave antigua
        zeroize1(session.fernet_key)
```

Listing 5: Forward secrecy al cerrar sesión

Gestión Segura de Memoria

- **Memory Locking**: `mlock()` para evitar swap
- **Zeroización**: Sobrescritura de claves con `zeroize1()`
- **Limpieza inmediata**: `del` tras uso de variables sensibles
- **Encriptación en memoria**: Base de datos siempre encriptada
- **Desencriptación bajo demanda**: Solo durante operaciones

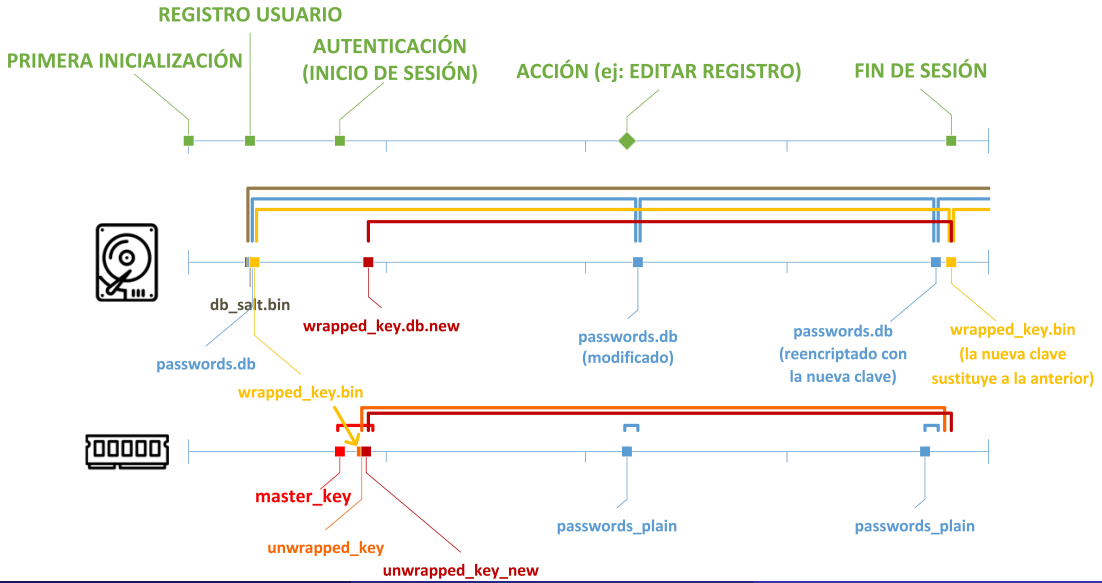
Gestión Segura de Memoria

```
class Session:
        def __init__(self, fernet_key, user_id, ...):
                # Almacenar en bytearray (mutable para zeroizacion)
                self.fernet_key = bytearray(fernet_key)
                self.fernet_key_next = bytearray(Fernet.generate_key())
                try:
                        # Bloquear memoria (evitar swap)
                        mlock(self.fernet_key)
                        mlock(self.fernet_key_next)
                        self.key_locked = True
                except Exception as e:
                        print(f"WARNING: {e}")

        def clear_key(self):
                if self.key_locked:
                        munlock(self.fernet_key)
                        zeroize1(self.fernet_key)  # Sobrescribir con
                            ceros
                        self.fernet_key = None
```

Listing 6: Memory locking y zeroización

Ejemplo práctico gestión de Memoria



Limitaciones y Trabajo Futuro

Limitaciones Actuales

- Almacenamiento local únicamente
- Soporte exclusivo para DNIe español
- Sin sincronización entre dispositivos
- Dependencia de hardware específico (lector DNIe)

Posibles Mejoras

- Sincronización cifrada en nube
- Integración con otros dispositivos de seguridad física
- Compartición segura entre usuarios
- Exportación/importación de credenciales