Bezpieczeństwo Systemów Komputerowych: Raport Termin Końcowy

Zespół:

Imię	Filip	Michał
Nazwisko	Wesołowski	Pawiłojć
Numer Indeksu	193486	193159
Link repozytorium	https://github.com/IkeaSzark/BSK	

1. Część 1: Stworzenie aplikacji auxillary generującej parę kluczy.

Pierwszym etapem projektu, którym się zajęliśmy było stworzenie pomocniczej aplikacji generującej parę kluczy, które później zostaną wykorzystane do szyfrowania pliku .pdf w aplikacji głównej.

```python
def DetectUSB():
    #Znajdź USB
    drive_list = win32api.GetLogicalDriveStrings()
    drive_list = drive_list.split("\x00")[0:-1]  #Ostatni element to ""
    for letter in drive_list:
        if win32file.GetDriveType(letter) == win32file.DRIVE_REMOVABLE: #Sprawdź czy ostatni elemnt to USB
            print("Dysk USB to: " + str(letter))
            return letter
    return ""
```

Pierwszym etapem działania aplikacji Auxillary jest wykrycie urządzenia USB, które zostanie wykorzystane do zapisu zaszyfrowanego klucza głównego. W tym celu wykorzystywane są funkcje win32api oraz win32file w celu uzyskania listy logicznej dysków, a następnie sprawdzenia czy ostatni element listy jest dyskiem wysuwanym.

Następnym krokiem jest wygenerowanie 4096-bitowego klucza RSA. Na jego podstawie generujemy nasz klucz prywatny oraz klucz publiczny.

```python
def GenerateRSA():
    #Wygeneruj 4096 bitowy klucz RSA
    key = RSA.generate(4096)
    private_key = key.export_key()
    #with open("private.pem", "wb") as pub_file:
    #    pub_file.write(private_key)
    public_key = key.publickey().export_key()
    return private_key, public_key
```

Kolejną czynnością naszej aplikacji jest pobranie od użytkownika 256-bitowego kodu pin, który zostanie wykorzystany podczas szyfrowania klucza prywatnego.

```python
def GetUserPin():
    # Get PIN from user and ensure it's 32 bytes long
    while True:
        pin = getpass.getpass("Enter your PIN (will be used to encrypt your private key): ")
        if len(pin) < 8:
            print("PIN must be at least 8 characters long")
            continue

        # Convert PIN to 32 bytes using SHA-256 hash if it's too short
        if len(pin) < 32:
            from Crypto.Hash import SHA256
            pin_hash = SHA256.new(pin.encode()).digest()
            print(f"Note: Your PIN has been hashed to 32 bytes for encryption")
            return pin_hash
        elif len(pin) > 32:
            print("Warning: PIN is longer than 32 bytes, it will be truncated")
            return pin.encode()[:32]
        else:
            return pin.encode()
```

Na podstawie kodu pin szyfrujemy nasz klucz prywatny za pomocą szyfrowania AES. Wygenerowany ciphertext, nonce i tag eksportujemy, abyśmy mogli zapisać zaszyfrowany już klucz prywatny na urządzeniu USB.

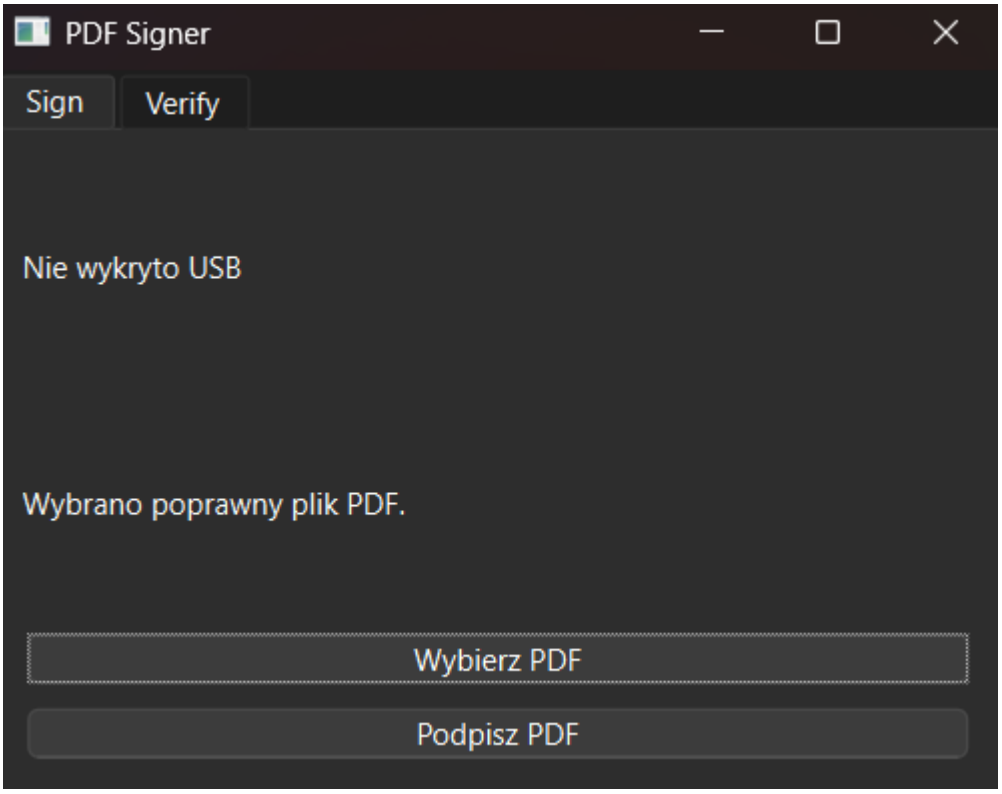
```python
def EncryptKey(pin, private_key):
    cipher = AES.new(pin, AES.MODE_OCB)
    ciphertext, tag = cipher.encrypt_and_digest(private_key)
    return cipher.nonce, tag, ciphertext

def ExportKey(nonce, tag, ciphertext):
    return base64.b64encode(nonce + tag + ciphertext).decode()
```

Wszystkie wygenerowane pliki zapisujemy. Klucz publiczny oraz pin (w celach testowych!) są zapisywane binarnie na dysku komputera. Zaszyfrowany klucz prywatny natomiast zapisywany jest automatycznie na urządzeniu USB.

```python
def SaveFiles(pin, exported_key, public_key, USB):
    # Zapis kluczy do plików
    with open("public.pem", "wb") as pub_file:
        pub_file.write(public_key)

    with open(USB+"encrypted_private.pem", "w") as priv_file:
        priv_file.write(exported_key)

    # Zapis PIN-u do pliku (UWAGA: PIN musi być przechowywany bezpiecznie!)
    with open("pin.bin", "wb") as pin_file:
        pin_file.write(pin)
```

2. Część 2: Stworzenie aplikacji podpisującej .pdf

Drugą częścią projektu było zaprojektowanie aplikacji głównej, która na podstawie wygenerowanych kluczy podpisze plik .pdf oraz zweryfikuje poprawność podpisu. Aplikacja napisana została w języku Python. Do interfejsu graficznego wykorzystana została biblioteka PyQt6 pozwalająca na tworzenie aplikacji okienkowych.



Główne okno aplikacji informuje użytkownika czy został wykryty automatycznie nośnik USB, który przechowuje klucz prywatny potrzebny do podpisu pliku .pdf. Pozwala również na weryfikację, czy wybrany przez użytkownika plik jest rzeczywiście plikiem .pdf.

```python
def DetectUSB(self):
    # Znajdź USB
    drive_list = win32api.GetLogicalDriveStrings()
    drive_list = drive_list.split("\x00")[0:-1]  # Ostatni element to ""
    for letter in drive_list:
        if win32file.GetDriveType(letter) == win32file.DRIVE_REMOVABLE:  # Sprawdź czy ostatni elemnt to USB
            print("Dysk USB to: " + str(letter))
            self._USB_letter = letter
            self._USB_detected = True
            return
    self._USB_detected = False
    self._USB_letter = ""
    return
```

```python
def open_file_dialog(self):
    file_dialog = QFileDialog(self)
    file_dialog.setWindowTitle("Open PDF")
    file_dialog.setFileMode(QFileDialog.FileMode.ExistingFile)
    file_dialog.setViewMode(QFileDialog.ViewMode.Detail)

    if file_dialog.exec():
        selected_file = file_dialog.selectedFiles()[0]
        if selected_file.lower().endswith(".pdf"):
            self.result_label.setText("Wybrano poprawny plik PDF.")
            self._PDF_file = selected_file
        else:
            self.result_label.setText("Wybrany plik nie jest plikiem PDF.")
```

Następnym krokiem aplikacji jest podpisanie pliku .pdf

```python
def sign_file(self):
    if not self._PDF_file:
        self.result_label.setText("Please select a PDF file first.")
        return

    if not self._USB_detected:
        self.result_label.setText("USB drive with private key not found.")
        return
```

Najpierw aplikacja weryfikuje czy wybrane ścieżki do pliku .pdf oraz czy poprawnie wykryto urządzenie USB.

```
try:
    # Get PIN from user
    from getpass import getpass
    pin = getpass("Enter your PIN to decrypt the private key: ")

    # Load encrypted private key from USB
    with open(self._USB_letter + "encrypted_private.pem", "r") as f:
        encrypted_data = f.read()

    # Decrypt the private key
    private_key = self.decrypt_private_key(encrypted_data, pin.encode())
    if not private_key:
        self.result_label.setText("Wrong PIN or corrupted private key.")
        return

    # Create PDF signature
    signature = self.create_pdf_signature(private_key, self._PDF_file)

    # Save signed PDF
    output_file = self._PDF_file.replace(".pdf", "_signed.pdf")
    with open(output_file, "wb") as f:
        f.write(signature)

    self.result_label.setText(f"PDF successfully signed and saved as: {output_file}")
```

Aplikacja pobiera od użytkownika jego numer pin i automatycznie pobiera z urządzenia .pdf zaszyfrowany klucz prywatny. Następnie na podstawie pinu przeprowadzana jest dekrypcja za pomocą algorytmu AES.

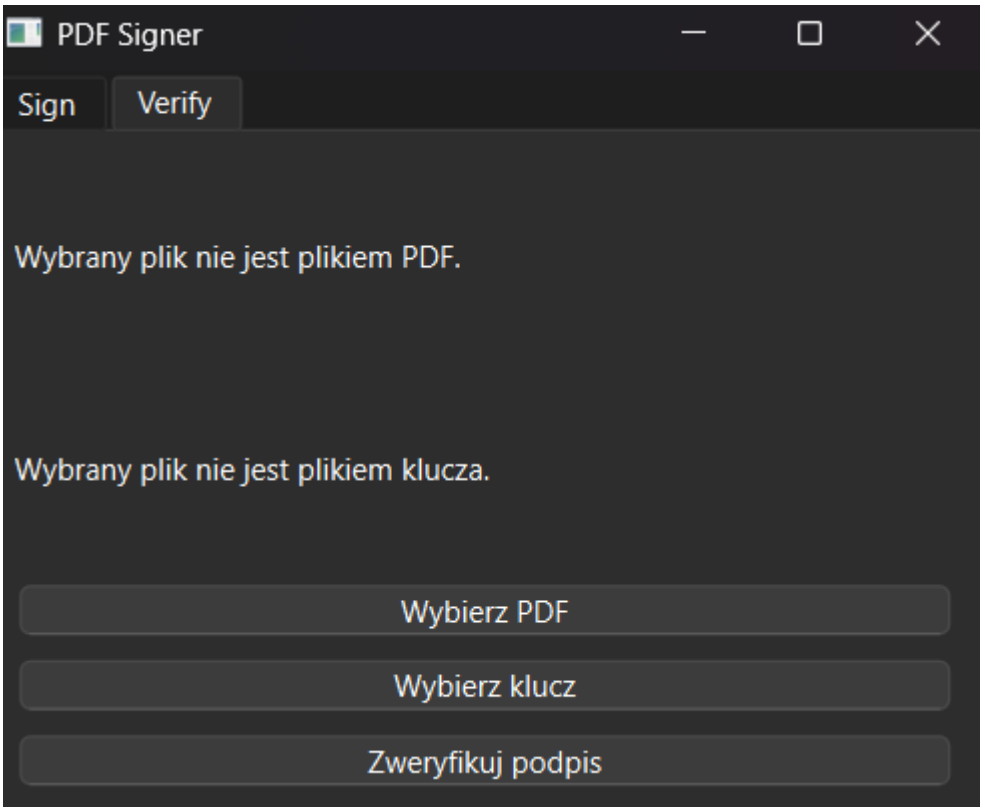
```python
def decrypt_private_key(self, encrypted_data, pin):
    try:
        import base64
        from Crypto.Cipher import AES

        # Prepare the key (hash if needed)
        if len(pin) < 32:
            pin = hashlib.sha256(pin).digest()
        elif len(pin) > 32:
            pin = pin[:32]

        # Decode and split the encrypted data
        encrypted_bytes = base64.b64decode(encrypted_data)
        nonce = encrypted_bytes[:15]  # OCB uses 15-byte nonce
        tag = encrypted_bytes[15:31]  # Tag is 16 bytes
        ciphertext = encrypted_bytes[31:]

        # Decrypt
        cipher = AES.new(pin, AES.MODE_OCB, nonce=nonce)
        return cipher.decrypt_and_verify(ciphertext, tag)
    except:
        return None
```

Kolejnym krokiem jest wygenerowanie podpisu i utworzenie nowego, podpisanego pliku .pdf.

```python
def create_pdf_signature(self, private_key, pdf_path):
    """Sign PDF content without affecting verifiability"""
    try:
        # Read original PDF
        with open(pdf_path, "rb") as f:
            original_pdf = PdfReader(f)
            writer = PdfWriter()

            # Copy all pages to preserve content exactly
            for page in original_pdf.pages:
                writer.add_page(page)

            # Create hash of the original content
            content_buffer = io.BytesIO()
            writer.write(content_buffer)
            content_to_sign = content_buffer.getvalue()
            pdf_hash = SHA256.new(content_to_sign)

            # Sign the hash
            rsa_key = RSA.import_key(private_key)
            signer = pkcs1_15.new(rsa_key)
            signature = signer.sign(pdf_hash)

            # Store signature in document info (doesn't affect content hash)
            writer.add_metadata({
                '/Signature': base64.b64encode(signature).decode('utf-8'),
                '/SigningDate': datetime.datetime.now().isoformat()
            })

            # Write signed PDF
            output_buffer = io.BytesIO()
            writer.write(output_buffer)
            return output_buffer.getvalue()
```

Wykorzystywany są do tego algorytm SHA256. W metadanych pliku .pdf zapisywane są informacje o dacie oraz podpisie.

Druga zakładka aplikacji pozwala na wybranie pliku .pdf oraz pliku z naszym kluczem publicznym i zweryfikowanie podpisu.




```python
def open_pdf_dialog(self):
    file_dialog = QFileDialog(self)
    file_dialog.setWindowTitle("Open PDF")
    file_dialog.setFileMode(QFileDialog.FileMode.ExistingFile)
    file_dialog.setViewMode(QFileDialog.ViewMode.Detail)

    if file_dialog.exec():
        selected_file = file_dialog.selectedFiles()[0]
        if selected_file.lower().endswith(".pdf"):
            self.result_label.setText("Wybrano poprawny plik PDF.")
            self._PDF_file = selected_file
        else:
            self.result_label.setText("Wybrany plik nie jest plikiem PDF.")
```

```python
def open_key_dialog(self):
    file_dialog = QFileDialog(self)
    file_dialog.setWindowTitle("Open Public Key")
    file_dialog.setFileMode(QFileDialog.FileMode.ExistingFile)
    file_dialog.setViewMode(QFileDialog.ViewMode.Detail)

    if file_dialog.exec():
        selected_file = file_dialog.selectedFiles()[0]
        if selected_file.lower().endswith(".pem"):
            self.result_label2.setText("Wybrano poprawny plik klucza.")
        else:
            self.result_label2.setText("Wybrany plik nie jest plikiem klucza.")
```

Aplikacja umożliwia wybranie plików klucza oraz .pdf za pomocą okna dialogowego bezpośrednio z dysku użytkownika.

```
def verify_signature(self):
    if not self._PDF_file:
        self.result_label.setText("Please select a PDF file first.")
        return

    if not self._Key_file:
        self.result_label.setText("Please select a public key file first.")
        return

    try:
        # Load public key
        with open(self._Key_file, "rb") as f:
            public_key_data = f.read()

        # Load the PDF
        with open(self._PDF_file, "rb") as f:
            pdf_content = f.read()

        # Verify the signature
        is_valid = self.verify_pdf_signature(public_key_data, pdf_content)

        if is_valid:
            self.result_label.setText("Signature is VALID")
            self.result_label.setStyleSheet("color: green")
        else:
            self.result_label.setText("Signature is INVALID")
            self.result_label.setStyleSheet("color: red")

    except Exception as e:
        self.result_label.setText(f"Error during verification: {str(e)}")
        self.result_label.setStyleSheet("color: red")
```

Następnie pliki te są weryfikowane i przeprowadzana zostaje operacja weryfikacji podpisu.

```python
def verify_pdf_signature(self, public_key_data, pdf_content):
    """Verify PDF signature while ignoring the signature metadata"""
    try:
        # Read PDF
        pdf_reader = PdfReader(io.BytesIO(pdf_content))

        # Get signature from metadata
        if not hasattr(pdf_reader, 'metadata') or not pdf_reader.metadata:
            return False

        signature_b64 = pdf_reader.metadata.get('/Signature', '')
        if not signature_b64:
            return False

        signature = base64.b64decode(signature_b64)

        # Reconstruct original content (without signature metadata)
        writer = PdfWriter()
        for page in pdf_reader.pages:
            writer.add_page(page)

        content_buffer = io.BytesIO()
        writer.write(content_buffer)
        content_to_verify = content_buffer.getvalue()

        # Verify signature
        pdf_hash = SHA256.new(content_to_verify)
        rsa_key = RSA.import_key(public_key_data)
        verifier = pkcs1_15.new(rsa_key)
        verifier.verify(pdf_hash, signature)

        return True

    except (ValueError, TypeError) as e:
        print(f"Verification failed: {str(e)}")
        return False
    except Exception as e:
        print(f"Verification error: {str(e)}")
        return False
```

Podczas weryfikacji pobierany jest podpis z metadanych pliku oraz zawartość pliku do zweryfikowania. Weryfikowanie zawartości ignoruje występujące metadane. Na koniec klucz publiczny zostaje

zaimportowany, odczytany i porównany z pod w celach ostatecznej weryfikacji dokumentu.

3. Zaimplementowane funkcjonalności

- Wygenerowanie 4096 bitowego klucza RSA,
- Wygenerowanie klucza prywatnego oraz publicznego,
- Automatyczne wykrycie urządzenia USB,
- Pobranie od użytkownika PIN'u o długości 256,
- Zaszyfrowanie klucza prywatnego za pomocą algorytmu AES na podstawie PIN'u,
- Bezpieczny zapis klucza prywatnego oraz jawny zapis klucza publicznego,
- Aplikacja okienkowa pozwalająca na wybranie pliku .pdf oraz pliku klucza z dysku za pomocą dialogu,
- Automatyczne odczytanie klucza prywatnego z dysku,
- Dekrypcja klucza prywatnego,
- Podpis wybranego pliku .pdf,
- Weryfikacja podpisanego pliku .pdf na podstawie klucza publicznego.