Embedded Security
Prof. Dr. Pelzl, Jan

Name : Pinak Ganatra
Email : pinak.ganatra@stud.hshl.de
Matrikel Nr. : 1210339

Datum : 06.07.2024

1. Introduction

This project presents a comprehensive smartcard-based security system designed for embedded systems. It integrates three key components: a Micropayment System, an Access Control System, and a Secure File System. By leveraging the security features of smartcards, this project demonstrates how to implement robust, multi-faceted security measures in various applications. The system utilizes advanced cryptographic techniques, including RSA for asymmetric encryption and digital signatures, AES for symmetric encryption, and implements two-factor authentication to ensure high levels of security.

2. Micro Payment System

The Micro Payment System is a crucial component of the smartcard-based security project, designed to facilitate secure financial transactions using smartcards in embedded systems. This system allows users to add funds, make payments, and check balances, all while ensuring the security and integrity of these operations.

2.1 Code Structure:

The MicroPayment System is implemented in the `MicroPaymentSystem` class, which contains several key methods:

I. Initialization and Configuration:
   A. `__init__()`: Initializes the system, sets up logging, and loads configuration.
   B. `load_config()`: Loads system configuration from a JSON file, with fallback to default values.

II. Smartcard Interaction:
   A. `connect_reader()`: Establishes a connection with the smartcard reader.
   B. `wait_for_card()`: Waits for a smartcard to be presented to the reader.
   C. `read_card()`: Reads the UID (Unique Identifier) from the presented smartcard.

III. Authentication and Security:
   A. `authenticate()`: Authenticates access to a specific block on the smartcard.
   B. `authenticated_operation()`: A context manager ensuring operations are performed only after authentication.

IV. Financial Operations:
   A. `read_balance()`: Reads the current balance from the smartcard.
   B. `write_balance()`: Updates the balance on the smartcard.
   C. `add_funds()`: Increases the balance on the smartcard.
   D. `make_payment()`: Decreases the balance on the smartcard.

V. Logging and Audit:
   A. `log_transaction()`: Records details of each transaction for auditing purposes.

2.1.1 Important Code Snippets:

I. **Authentication:** This method sends an authentication command to the smartcard and checks for a successful response.

```python
def authenticate(self, block):
    try:
        auth = [0xFF, 0x86, 0x00, 0x00, 0x05, 0x01, 0x00, block, 0x60, 0x00]
        response, sw1, sw2 = self.connection.transmit(auth)
        if sw1 == 0x90 and sw2 == 0x00:
            return True
        else:
            self.logger.warning(f"Authentication failed. SW1: {sw1:02X}, SW2: {sw2:02X}")
            return False
    except Exception as e:
        self.logger.error(f"Error during authentication: {e}")
        return False
```

II. **Secure Balance Operations:** This method reads the balance from the smartcard, ensuring that the operation is authenticated.

```python
def read_balance(self):
    try:
        with self.authenticated_operation(self.balance_block):
            read_command = [0xFF, 0xB0, 0x00, self.balance_block, 0x10]
            data, sw1, sw2 = self.connection.transmit(read_command)
            if sw1 == 0x90 and sw2 == 0x00:
                return struct.unpack('>I', bytes(data[:4]))[0]
            else:
                self.logger.warning(f"Error reading balance. SW1: {sw1:02X}, SW2: {sw2:02X}")
                return None
    except Exception as e:
        self.logger.error(f"Error reading balance: {e}")
        return None
```

2.2 Authentication Mechanism

The Micro Payment System uses a simple challenge-response authentication mechanism. Before any operation that reads or writes to the smartcard, the system must authenticate itself to the card. This is done by sending a specific command sequence to the card and verifying the response. The `authenticated_operation` context manager ensures that every critical operation (like reading or writing the balance) is performed only after successful authentication:

```python
@contextmanager
def authenticated_operation(self, block):
    if self.authenticate(block):
        try:
            yield
        finally:
            pass  # No specific cleanup needed after authentication
    else:
        raise Exception("Authentication failed")
```

2.3 Characteristics of the Secure Payment System:

I. Data Integrity: By storing the balance on the smartcard itself, the system ensures that the financial data is always with the user and not vulnerable to remote attacks.
II. Authentication: Each operation requires authentication, preventing unauthorized access to the stored balance.
III. Secure Communication: The system uses encrypted communication between the reader and the smartcard, protecting against eavesdropping.
IV. Audit Trail: All transactions are logged, providing a clear audit trail for dispute resolution or detecting fraudulent activities.
V. Offline Capability: The system can operate without a constant network connection, making it suitable for various embedded applications.
VI. Tamper Resistance: Smartcards are designed to be tamper-resistant, providing an additional layer of physical security.
VII. Modular Design: The system's modular design allows for easy integration with other components of the overall security system.

This Micro Payment System contributes to the overall project by demonstrating how smartcards can be used to implement secure financial transactions in embedded systems. It showcases the integration of cryptographic operations, secure data storage, and authentication mechanisms in a practical, real-world application.

3. Access Control System Description

The Access Control System is a critical component of the smartcard-based security project, designed to manage and control physical access to various areas using smartcards. This system allows for user registration, authentication, and access verification, ensuring that only authorized individuals can enter specific areas.

3.1 Code Structure

The Access Control System is implemented in the `SecuritySystem` class, which contains several key methods:

1. Initialization and Configuration:
    - `__init__()`: Initializes the system, sets up logging, and loads configuration and user data.
    - `load_config()`: Loads system configuration from a JSON file, with fallback to default values.
    - `load_users()`: Loads user data from a JSON file.
2. Smartcard Interaction:
    - `connect_reader()`: Establishes a connection with the smartcard reader.
    - `wait_for_card()`: Waits for a smartcard to be presented to the reader.
    - `read_card()`: Reads the UID (Unique Identifier) from the presented smartcard.
3. Authentication and Security:
    - `challenge_response_auth()`: Implements a challenge-response authentication protocol using RSA.
    - `load_or_generate_key()`: Loads or generates an RSA key pair for the system.
4. Access Control Operations:
    - `check_access()`: Verifies if a user has access to a specific area.
    - `log_access()`: Records access attempts for auditing purposes.
    - `add_user()`: Registers a new user in the system.
    - `remove_user()`: Removes a user from the system.

3.1.1 Important Code Snippets

I. Challenge-Response Authentication: This method implements a challenge-response protocol using RSA for secure authentication.

```
def challenge_response_auth(self, uid):
    if uid not in self.users:
        return False
    challenge = os.urandom(32)
    signature = self.private_key.sign(
        challenge,
        padding.PSS(
            mgf=padding.MGF1(hashes.SHA256()),
            salt_length=padding.PSS.MAX_LENGTH
        ),
        hashes.SHA256()
    )
    try:
        self.private_key.public_key().verify(
            signature,
            challenge,
            padding.PSS(
                mgf=padding.MGF1(hashes.SHA256()),
                salt_length=padding.PSS.MAX_LENGTH
            ),
            hashes.SHA256()
        )
        return True
    except:
        return False
```

II. Access Control: This method checks if a user has access to a specific area.

```python
def check_access(self, uid, area):
    if uid not in self.users:
        return False
    return area in self.users[uid]['access_areas']
```

3.2 Authentication Mechanism

The Access Control System uses a robust challenge-response authentication mechanism based on RSA cryptography. The process works as follows:

1. The system generates a random challenge.
2. The challenge is signed using the system's private key.
3. The signature is then verified using the public key.
4. If the verification succeeds, the authentication is considered successful.

This method provides strong security against replay attacks and ensures that only holders of valid smartcards can gain access.

3.3 Characteristics of the Secure Access System

1. Strong Authentication: The challenge-response mechanism provides a high level of security, protecting against various types of attacks.
2. Flexible Access Control: The system allows for fine-grained control over which users can access specific areas.
3. User Management: The system includes functions for adding and removing users, making it easy to manage access rights.
4. Audit Trail: All access attempts are logged, providing a clear record for security audits and investigations.
5. Cryptographic Security: The use of RSA cryptography ensures that the authentication process is mathematically secure.
6. Scalability: The system can easily accommodate a large number of users and areas.
7. Integration: The modular design allows for easy integration with other security systems, such as alarms or surveillance cameras.
8. Offline Capability: The system can operate without a constant network connection, making it suitable for various embedded applications.

This Access Control System contributes to the overall project by demonstrating how smartcards can be used to implement secure physical access control in embedded systems. It showcases the integration of advanced cryptographic operations, user management, and access control mechanisms in a practical, real-world application. The system provides a crucial layer of security for controlling entry and exit in sensitive areas, complementing the financial security provided by the Micro Payment System.

4. Secure File System

The Secure File System is a sophisticated component of the smartcard-based security project, designed to provide secure storage and access to sensitive files using smartcards. This system allows users to encrypt and decrypt files, ensuring that data remains protected even if the storage medium is compromised.

4.1 Code Structure

The Secure File System is implemented in the `SecureFileSystem` class, which contains several key methods:

1. Initialization and Configuration:
    - `__init__()`: Initializes the system, sets up logging, and loads configuration and user data.
    - `load_config()`: Loads system configuration from a JSON file, with fallback to default values.
    - `load_users()`: Loads user data from a JSON file.
2. Smartcard Interaction:
    - `connect_reader()`: Establishes a connection with the smartcard reader.
    - `wait_for_card()`: Waits for a smartcard to be presented to the reader.
    - `read_card()`: Reads the UID (Unique Identifier) from the presented smartcard.
3. User Management and Authentication:
    - `authenticate_user()`: Implements two-factor authentication using smartcard and PIN.
    - `register_user()`: Registers a new user with a smartcard, PIN, and generates an RSA key pair.
4. Cryptographic Operations:
    - `generate_key()`: Generates a random symmetric key for AES encryption.
    - `encrypt_file()`: Encrypts a file using AES and protects the AES key with RSA.
    - `decrypt_file()`: Decrypts a file using the user's RSA private key and AES.

4.1.1 Important Code Snippets

I. Two-Factor Authentication: This method implements two-factor authentication using the smartcard (something one have) and a PIN (something one know).

```python
def authenticate_user(self, uid):
    if uid not in self.users:
        print("Card not recognized. Please register first.")
        return False

    pin_attempt = input("Enter your PIN: ")
    if pin_attempt == self.users[uid]['pin']:
        print("Authentication successful.")
        return True
    else:
        print("Incorrect PIN.")
        return False
```

II. File Encryption: This method encrypts a file using AES in CBC mode and protects the AES key using RSA encryption.

```python
def encrypt_file(self, uid, filename):
    if not self.authenticate_user(uid):
        return

    file_path = os.path.join(self.secure_folder, filename)
    if not os.path.exists(file_path):
        print(f"File {filename} not found in secure folder.")
        return

    with open(file_path, 'rb') as file:
        data = file.read()

    # Generate a random symmetric key
    symmetric_key = self.generate_key()

    # Encrypt the file data with AES
    iv = os.urandom(16)
    cipher = Cipher(algorithms.AES(symmetric_key), modes.CBC(iv), backend=default_backend())
    encryptor = cipher.encryptor()
    padder = symmetric_padding.PKCS7(128).padder()
    padded_data = padder.update(data) + padder.finalize()
    encrypted_data = encryptor.update(padded_data) + encryptor.finalize()
```

```python
# Encrypt the symmetric key with RSA
public_key = serialization.load_pem_public_key(
    self.users[uid]['public_key'].encode('utf-8'),
    backend=default_backend()
)
encrypted_symmetric_key = public_key.encrypt(
    symmetric_key,
    asymmetric_padding.OAEP(
        mgf=asymmetric_padding.MGF1(algorithm=hashes.SHA256()),
        algorithm=hashes.SHA256(),
        label=None
    )
)

# Write the encrypted data and key to file
with open(file_path + '.encrypted', 'wb') as file:
    file.write(len(encrypted_symmetric_key).to_bytes(4, byteorder='big'))
    file.write(encrypted_symmetric_key)
    file.write(iv)
    file.write(encrypted_data)

os.remove(file_path)
print(f"File {filename} encrypted successfully.")
```

4.2 Authentication Mechanism

The Secure File System uses a two-factor authentication mechanism:

1. Smartcard Presence: The system verifies the presence of a registered smartcard.
2. PIN Verification: The user must enter a correct PIN associated with the smartcard.

This two-factor approach significantly enhances security by requiring both possession of the smartcard and knowledge of the PIN.

4.2.1 Encryption Mechanism

The system employs a hybrid encryption scheme:

1. AES (Advanced Encryption Standard) for file content encryption, providing fast and secure symmetric encryption.
2. RSA for encrypting the AES key, allowing secure key management and distribution.

4.3 Characteristics of the file Securing System

1. Strong Two-Factor Authentication: Combines smartcard (possession factor) with PIN (knowledge factor) for robust user authentication.
2. Hybrid Encryption: Utilizes the speed of symmetric encryption (AES) for file content and the security of asymmetric encryption (RSA) for key protection.
3. Data Confidentiality: Ensures that files remain encrypted and unreadable without proper authentication and decryption keys.
4. Key Management: RSA key pairs are generated and stored securely for each user, facilitating secure key distribution.
5. File Integrity: Encryption protects against unauthorized modifications to the file content.
6. User-Specific Access: Each user has their own encryption keys, ensuring that files can only be accessed by authorized individuals.
7. Offline Security: Files remain secure even when the system is offline or the storage medium is physically accessed by unauthorized parties.
8. Scalability: The system can handle multiple users and a large number of files efficiently.

This Secure File System contributes to the overall project by demonstrating how smartcards can be used to implement secure file storage and access in embedded systems. It showcases the integration of advanced cryptographic operations, including symmetric and asymmetric encryption, with smartcard-based authentication. This system provides a crucial layer of data protection, complementing the financial security of the Micro Payment System and the physical security of the Access Control System. Together, these three components create a comprehensive security solution for embedded systems, addressing financial, physical, and data security needs.

5. Conclusion

This smartcard-based security project for embedded systems addresses key security challenges in modern digital environments by integrating a Micro Payment System, an Access Control System, and a Secure File System. These components demonstrate the versatility and robustness of smartcard technology in providing comprehensive security solutions. The Micro Payment System ensures secure financial transactions, the Access Control System manages physical security with advanced cryptographic techniques, and the Secure File System protects sensitive data through hybrid encryption and two-factor authentication. Key security principles implemented include multi-factor authentication, encryption, secure key management, audit logging, and offline security capabilities, creating a holistic security ecosystem for financial, physical, and data protection needs in embedded systems.

5.1 Future Development

Several avenues for further development include:

- **Biometric Integration:** Add biometric authentication (e.g., fingerprint or facial recognition) for enhanced security.
- **Blockchain Integration:** Use blockchain for the Micro Payment System to provide immutable transaction records and transparency.
- **Network Security:** Develop secure communication protocols for network interactions, ensuring data integrity and confidentiality.
- **Machine Learning for Anomaly Detection:** Implement ML algorithms to detect unusual system usage patterns and identify security breaches.
- **Mobile App Integration:** Create a companion mobile app for user convenience in accessing data and transaction history.
- **IoT Integration:** Extend the system for secure control and monitoring of smart home or industrial IoT devices.
- **Regulatory Compliance:** Ensure compliance with data protection regulations (e.g., GDPR, CCPA) across different jurisdictions.
- **Scalability Enhancements:** Optimize the system for large-scale deployments using distributed systems concepts.
- **User Interface Improvements:** Develop a more intuitive and user-friendly interface for system administration and user interactions.

These enhancements will make the smartcard-based security system more powerful and versatile, capable of addressing emerging security challenges in embedded systems and IoT. The current modular implementation provides a strong foundation for these future developments, ensuring adaptability to new security requirements and technological advancements.

6. Citations

AI Tools Used	**What were AI Tools used for?**
Quillbot	Paraphrase, writing assistance
Claude AI	Research Assistance, information extraction from pdfs and images etc. Assist for code debugging