**1. 암호 모듈 개요**

* **모듈 이름 및 버전**: "ARIA-SHA512 암호 모듈 v1.0".
* **모듈 유형**: 소프트웨어 기반
* **기능적 개요**: ARIA 암호화 알고리즘을 사용한 데이터 암호화 및 SHA-512 해시 알고리즘을 사용한 데이터 무결성 확인 기능 제공

**2. 암호 알고리즘 및 키 관리 사용 암호 알고리즘**:

* + **ARIA**: 대칭 키 블록 암호 알고리즘으로, 128비트, 192비트, 256비트 키 크기를 지원
  + **SHA-512**: 512비트 해시 함수로, 무결성 확인과 디지털 서명을 위한 해시값을 생성
* **지원하는 암호화 모드**: ARIA를 CBC, CTR, GCM과 같은 운영 모드
* **키 생성 및 관리**:
  + **키 생성**: ARIA의 대칭 키는 모듈 내부에서 안전하게 생성되거나 외부에서 안전하게 입력
  + **키 저장**: 키는 휘발성 메모리에서만 저장되며, 모듈 종료 시 자동으로 삭제
  + **키 폐기**: 키는 사용 완료 후 안전하게 폐기

**3. 암호 모듈의 보안 경계**

**• 물리적 보안 경계**

암호 모듈은 소프트웨어 기반으로 설계되며, 실행되는 플랫폼의 보안 경계는 해당 소프트웨어가 설치된 컴퓨팅 환경(예: 서버, 워크스테이션)의 운영 체제와 하드웨어를 포함한다. 이 암호 모듈은 특정 하드웨어 장치 내에서 실행되지 않으므로, 물리적 보안 경계는 사용 중인 호스트 시스템의 물리적 접근 제어에 의존한다. 예를 들어, 호스트 장치의 물리적 보안은 잠금 장치, 접근 통제 시스템 등을 통해 유지된다. 모든 키와 민감한 정보는 휘발성 메모리 내에서 처리되며, 외부 저장소에는 저장되지 않는다.

**• 논리적 보안 경계**

* 데이터 입력 경로: ARIA 알고리즘으로 암호화할 평문 데이터와 SHA-512로 처리할 데이터를 입력하는 인터페이스를 통해 데이터가 모듈로 들어옵니다.
* 데이터 출력 경로: 암호화된 데이터(ARIA)와 해시 값(SHA-512) 결과는 모듈의 데이터 출력 인터페이스를 통해 외부로 전달됩니다.
* 제어 입력 경로: 암호화 알고리즘의 선택, 키 입력 및 설정, 해시 처리 명령 등이 제어 입력 인터페이스를 통해 전달됩니다.
* 상태 출력 경로: 암호 모듈의 정상 동작, 오류 상태 등은 상태 출력 인터페이스를 통해 제공되며, 이를 통해 모듈의 현재 상태를 확인할 수 있습니다.

**4. 인터페이스 사양 데이터 입력 인터페이스**:

평문 데이터(Plaintext)는 ARIA 암호화 알고리즘에 전달되며, 대칭 키 암호화를 위해 키도 함께 입력됩니다.

해시할 데이터는 SHA-512 해시 함수에 전달되며, 파일, 메시지 또는 데이터 블록과 같은 정보를 입력으로 받습니다.

키 설정 및 키 입력 또한 데이터 입력 인터페이스를 통해 이루어집니다.

* **데이터 출력 인터페이스**:

ARIA로 암호화된 데이터는 평문 데이터를 받아 대칭 키로 암호화한 후 암호문을 출력합니다.

SHA-512 해시 값은 해시 처리된 결과를 출력하며, 무결성 검증 또는 디지털 서명에 사용될 수 있습니다.

출력된 데이터는 별도의 통신 경로나 외부 저장소로 전달됩니다.

* **제어 입력 인터페이스**:

사용자는 ARIA 알고리즘의 운영 모드(CBC, GCM 등)를 선택할 수 있으며, 이 명령은 제어 입력 인터페이스를 통해 모듈에 전달됩니다.

SHA-512 알고리즘으로 해시 처리를 요청하거나, 모듈 내에서 특정 보안 정책을 설정하는 명령을 입력할 수 있습니다.

모듈의 초기화, 인증 과정, 키 교체 등 관리 명령도 제어 입력 인터페이스를 통해 처리됩니다.

* **상태 출력 인터페이스**:

암호화 및 해시 처리가 정상적으로 완료되었음을 알리는 "정상 동작" 메시지가 출력됩니다.

잘못된 키 입력이나 데이터 손상 등의 문제가 발생했을 경우, "오류 발생" 상태가 출력됩니다.

암호 모듈이 초기화 상태에 있거나 비활성화된 상태일 때 "대기 중" 상태를 출력할 수 있습니다.

* **5. 운영 환경**
* **플랫폼 및 운영체제**:

**Windows 기반 환경**: 암호 모듈은 Windows 10 또는 Windows Server 2019 환경에서 실행되며, Windows의 네이티브 보안 기능인 BitLocker와 함께 사용할 수 있습니다.

* **운영 환경의 보안 요건**:

운영체제 보호: 암호 모듈이 설치된 운영체제는 파일 시스템 암호화, 접근 통제 리스트(ACL), 사용자 인증 관리 등의 보안 기능을 제공해야 합니다. 이를 통해 암호 모듈에 대한 무단 접근을 방지할 수 있습니다.

메모리 보호: 민감한 데이터와 암호 키는 운영체제의 메모리 보호 기능을 통해 안전하게 관리되며, 휘발성 메모리에만 일시적으로 저장됩니다. 운영체제는 메모리 접근을 제한하여 공격자가 암호화 키를 추출하지 못하도록 보호합니다.

물리적 보안 요구사항: 임베디드 장치의 경우, 물리적 보안 조치를 통해 암호 모듈이 탑재된 하드웨어가 무단으로 조작되거나 변조되는 것을 방지합니다. 하드웨어 레벨의 변조 방지 기술(예: 감지 센서)이 필요할 수 있습니다.

**6. 무결성 보호 및 자체 테스트 모듈 무결성 보호**: 모듈 실행 시, 모듈 파일 전체를 SHA-512 해시로 검증하여 변조 여부를 확인합니다. 무결성 검사가 실패하면 모듈은 '변조 감지' 상태를 출력하고 추가 동작을 중지합니다.

* **자체 테스트**:

알고리즘 정확성 테스트: ARIA 암호화 및 복호화 알고리즘의 정확성을 검증하기 위해 사전 정의된 입력값에 대한 출력값을 비교합니다.

SHA-512 테스트: 해시 값 계산이 정확히 이루어지는지 확인한다.

난수 발생기 테스트: 키 생성에 사용되는 난수 발생기가 기대한 대로 작동하는지 확인한다.

키 무결성 테스트: 저장된 암호화 키의 무결성을 확인하여 손상 여부를 탐지한다.

**7. 키 관리 및 보호**

* **키 생성**: 암호 모듈은 CSPRNG를 사용해 암호 키를 생성하며, 생성된 키는 휘발성 메모리에서만 일시적으로 저장됩니다. 생성된 키는 사용자에게 직접적으로 노출되지 않는다.
* **키 배포**: 네트워크를 통해 키를 전송할 때는 TLS 또는 IPsec과 같은 안전한 프로토콜을 사용하여 전달하며, 키는 공개 키 암호화 방식으로 암호화되어 전송된다.
* **키 저장 및 폐기**: 작업이 완료된 후 휘발성 메모리에 저장된 모든 키는 자동으로 덮어쓰기로 삭제되며, 삭제 절차가 완료되었을 때 상태 출력 인터페이스를 통해 삭제 성공 여부를 알린다.

**8. 오류 처리**

암호화 오류: 입력된 데이터의 길이가 암호화 규칙에 맞지 않거나 잘못된 키가 입력되었을 경우, 모듈은 오류 메시지를 출력하고, 데이터 처리 없이 안전한 상태로 전환됩니다.

무결성 검사 실패: 무결성 검사가 실패하면, 모듈은 데이터를 즉시 폐기하고 '오류 상태' 메시지를 출력하여 운영자가 조치를 취할 수 있도록 합니다.

**9. 감사 및 로깅**

성공적인 암호화/복호화 작업: 각각의 암호화 및 복호화 작업에 대해 시간, 사용된 키 ID, 입력 데이터 길이 등이 기록됩니다.

오류 발생 이벤트: 암호화 또는 해시 처리 실패 시 해당 오류의 세부사항이 로그에 기록됩니다.

변조 감지 이벤트: 모듈의 무결성 검사 실패 시, 해당 이벤트와 관련된 세부사항이 기록됩니다.

**10. 보안 등급 및 인증**

KCMVP 1단계 보안 등급을 충족하는 암호모듈

**11. 물리적 보안**

**암호모듈 인터페이스**

**-구체적인 프로그램 흐름을 정해서 그걸 표현하면 되는거 같음**

**기타 공격에 대한 대응**

**난독화 방법**

**난독화는 소스 코드나 바이트 코드를 고의적으로 복잡하게 만들어 역공학이나 분석을 어렵게 하는 기법이다. 아래는 대표적인 난독화 방법들이다.**

1. **이름 변환 (Identifier Renaming) 클래스, 메서드, 변수 등의 이름을 의미가 없는 이름으로 바꾸는 방법이다. 사람은 직관적인 이름을 통해 코드의 기능을 파악하지만, 이름을 난독화하면 코드의 목적을 알아보기 어렵게 된다. 예를 들어, 변수 userAge를 a1, methodName을 m2로 변환하면 코드 이해가 훨씬 어려워진다.**

**java**

**코드 복사**

**// 원래 코드**

**int userAge = 30;**

**String methodName = "getUserInfo";**

**// 난독화된 코드**

**int a1 = 30;**

**String m2 = "getUserInfo";**

1. **제어 흐름 변경 (Control Flow Obfuscation) 코드의 논리적인 흐름을 유지하면서도 흐름을 복잡하게 만드는 방법이다. 불필요한 조건문이나 루프를 추가하거나, 순서를 뒤섞어서 프로그램의 동작은 유지되지만 코드 분석이 복잡해진다.**

**java**

**코드 복사**

**// 원래 코드**

**if (isUserValid) {**

**processLogin();**

**}**

**// 난독화된 코드**

**if (isUserValid == true || isUserValid == false) {**

**if (isUserValid) {**

**processLogin();**

**}**

**}**

1. **문자열 암호화 (String Encryption) 코드 내에 포함된 문자열을 암호화하여 코드 상에서는 알아볼 수 없게 하고, 실행 시에만 복호화해서 사용하는 방법이다. 주로 중요한 정보나 메시지를 숨기기 위해 사용한다.**

**java**

**코드 복사**

**// 원래 코드**

**String secretMessage = "Important Data";**

**// 난독화된 코드 (암호화)**

**String encryptedMessage = "aGVsbG8gd29ybGQ="; // Base64로 암호화**

1. **클래스와 메서드 분할 (Class and Method Splitting) 하나의 큰 클래스나 메서드를 여러 개로 분리하여 코드를 분석하기 어렵게 만드는 방법이다. 이렇게 나누어진 클래스나 메서드들은 서로 복잡하게 연결되어 있어, 흐름을 파악하기 어려워진다.**
2. **데드 코드 추가 (Dead Code Insertion) 실행되지 않는 코드(데드 코드)를 일부러 삽입하여 코드 분석을 혼란스럽게 만드는 방법이다. 이 코드는 실제로 동작하지 않지만 코드 분석 도구가 이를 분석하려다 더 복잡한 분석 과정을 거치게 된다.**

**취약한 코드의 예시**

**취약한 코드는 보안상의 허점을 남겨 공격자가 이를 악용할 수 있는 코드다. 아래는 대표적인 취약한 코드의 예시와 설명이다.**

1. **하드코딩된 비밀번호 코드에 민감한 정보를 하드코딩하면, 공격자가 소스 코드를 분석하여 비밀번호를 쉽게 얻을 수 있다. 이는 매우 위험한 방식이다.**

**java**

**코드 복사**

**// 취약한 코드: 비밀번호가 하드코딩됨**

**String password = "123456";**

**해결 방법: 비밀번호나 민감한 정보는 환경 변수나 안전한 저장소에서 가져와야 한다.**

**java**

**코드 복사**

**// 개선된 코드: 비밀번호를 안전한 방식으로 저장 및 불러옴**

**String password = System.getenv("PASSWORD");**

1. **SQL 인젝션 공격에 취약한 코드 사용자 입력을 검증하지 않고 직접 SQL 쿼리로 사용할 경우, SQL 인젝션 공격에 취약해진다.**

**java**

**코드 복사**

**// 취약한 코드**

**String query = "SELECT \* FROM users WHERE username = '" + username + "' AND password = '" + password + "'";**

**공격자는 username 필드에 admin' --와 같은 값을 넣어 쿼리의 동작을 변경할 수 있다.**

**해결 방법: SQL 쿼리를 작성할 때는 준비된 문(statement)이나 파라미터 바인딩을 사용하여 입력값을 안전하게 처리한다.**

**java**

**코드 복사**

**// 개선된 코드**

**String query = "SELECT \* FROM users WHERE username = ? AND password = ?";**

**PreparedStatement pstmt = connection.prepareStatement(query);**

**pstmt.setString(1, username);**

**pstmt.setString(2, password);**

1. **버퍼 오버플로우 C/C++ 같은 언어에서는 사용자 입력이 예상보다 큰 경우, 메모리 범위를 초과하여 버퍼 오버플로우가 발생할 수 있다. 이는 악의적인 코드 실행으로 이어질 수 있다.**

**c**

**코드 복사**

**// 취약한 코드**

**char buffer[10];**

**strcpy(buffer, input); // input이 10자 이상이면 버퍼 오버플로우 발생**

**해결 방법: 입력값의 길이를 미리 확인하고, 적절한 버퍼 크기를 할당하거나 안전한 함수를 사용한다.**

**c**

**코드 복사**

**// 개선된 코드**

**char buffer[10];**

**strncpy(buffer, input, sizeof(buffer) - 1); // 버퍼의 크기를 초과하지 않도록 제한**

**buffer[sizeof(buffer) - 1] = '\0'; // null terminator 추가**

**이러한 취약한 코드들은 시스템의 보안에 큰 위협이 될 수 있기 때문에, 개발 과정에서 코드 리뷰와 보안 점검을 통해 발견하고 수정해야 한다.**