OWASP 시큐어 코딩 규칙 참고 가이드

- 번역 (2011 년 6 월): OWASP Korea 챕터
- OWASP Korea 연락처: 최훈진 이사 (gmanhj@ahnlab.com)

목차

소개

본 기술 문서는 소프트웨어 개발 생명주기의 한 부분으로 통합하여 사용할 수 있는 일반적인 소프트웨어 시큐어 코딩 규칙을 체크리스트 형태로 정의하고 있다. 이 규칙을 적용한다면 가장 일반적인 소프트웨어 취약점을 완화시킬 수 있다.

보안 사고와 관련된 비용을 따지지 않더라도, 소프트웨어 패키지가 완성된 후 보안 문제를 수정하는 것보다 안전한 소프트웨어를 개발하는 것이 비용이 적게 든다.

공격자의 초점이 애플리케이션 계층으로 지속적으로 이동함에 따라 주요 소프트웨어 자원 보호는 그 이전보다 중요해졌다. 2009 년 SANS 연구결과 [1] 에 따르면 애플리케이션 계층에 대한 공격이 인터넷에서 발견되는 전체 공격 시도의 60% 이상을 차지하고 있다.

본 가이드를 활용할 때, 개발팀은 보유한 소프트웨어 개발 생명주기와 팀원의 지식 수준의 성숙도에 대한 평가부터 시작해야 한다. 본 가이드는 개별 코딩 규칙을 구현하는 상세한 내용이 없으므로, 사전 지식이나 필요한 조언을 얻을 수 있는 충분한 자원(조언자)이 개발자에게 필요하다. 본 가이드는 보안 취약점과 공격코드에 대해 깊게 알고 있는 개발자가 없어도 코딩 요구사항으로 사용할 수 있는 코딩 규칙을 제공한다. 하지만, 개발팀은 전체 시스템의 설계와 구현 안전성을 검증할 수 있는 책임감과 충분한 교육, 도구 그리고 자원을 가져야 한다.

이탤릭으로 표기한 본 문서, 장별 제목, 그리고 문장의 중요 단어 목록은 부록 B 에서 제공한다.

안전한 소프트웨어 개발 프레임워크 구현 가이드라인은 본 문서의 범위에 포함되지 않는다. 하지만, 추가적으로 필요한 일반적인 사례 문서와 참고 문헌은 다음과 같다:

- 역할과 책임을 명확하게 정의
- 개발팀에게 충분한 소프트웨어 보안 교육을 제공
- 안전한 소프트웨어 개발 생명주기 준수
  - OWASP CLASP Project
- 시큐어 코딩 표준 확립
  - OWASP Development Guide Project
- 재사용 가능한 객체 라이브러리 제작
  - OWASP Enterprise Security API (ESAPI) Project
- 보안 통제의 효과성 점검
  - OWASP Application Security Verification Standard (ASVS) Project)
- RFP(the Request for Proposal)와 계약서에 보안 요구사항과 검증 방법론을 정의한 보안 아웃소싱 개발 규칙을 확립
  - OWASP Legal Project

소프트웨어 보안과 위험 원칙 개요

안전한 소프트웨어를 개발하려면 보안 원칙에 대한 기본적인 이해가 필요하다. 보안 원칙에 대한 상세한 설명은 본 가이드의 범위를 벗어나므로, 개괄적인 내용만을 제공한다.

소프트웨어 보안의 목표는 성공적으로 사업을 운영하기 위해 정보 자원의 *비밀성*, *무결성*, 그리고 *가용성*을 유지하는 것이다. 이 목표들은 *보안 통제 기능*을 구현하면서 달성할 수 있다. 본 가이드는 흔한 소프트웨어의 *취약점*을 *완화*시킬 수 있는 기술적 통제 항목에 중점을 둔다. 1 차적인 초점은 웹 애플리케이션과 웹 애플리케이션을 지원하는 인프라 스트럭처이지만, 가이드라인의 내용 대부분은 모든 소프트웨어 개발 플랫폼에 적용 가능하다.

소프트웨어의 안전성과 관련된 수용할 수 없는 위험으로부터 사업을 보호하려면, 위험의 의미를 이해할 필요가 있다. 위험은 비즈니스의 성공을 위협하는 요소들의 조합이다. 이는 개념적으로 다음과 같이 설명 가능하다: *위협 요소*는 *피해*를 주기위해 *악용*할 수 있는 *취약점*을 가진 *시스템*과 상호작용한다. 이 것은 추상적인 개념 같지만, 다음과 같은 방향으로 생각해보자: 자동차 도둑(위협 요소)이 잠기지 않은 문(취약점)을 찾기 위해 주차장을 따라 자동차(시스템)을 확인하면서 간다. 그리고 도둑이 하나 발견한다면, 자동차 문을 열고(악용) 자동차 안에 무엇이 있든 가져간다(피해). 이러한 모든 요소들이 안전한 소프트웨어 개발에 영향을 준다.

개발팀의 접근방법과 애플리케이션 공격자의 접근방법은 기본적으로 차이점이 있다. 개발팀은 일반적으로 애플리케이션의 정상적인 의도에 초점을 맞춰 접근한다. 즉 문서화된 기능 요구사항과 사용자 사례에 기반하여 특정 기능을 수행하는 애플리케이션을 디자인한다. 이와 달리 공격자는 애플리케이션이 할 수 있는 것과 "구체적으로 거부되지 않은 행위는 허용된다"라는 원칙에 따라 동작하는 것에 더 많은 관심을 가지고 있다. 이 문제를 해결하려면, 소프트웨어 생명주기의 초기 단계에서부터 추가적인 요소가 포함되어야 한다. 추가적으로 필요한 요소란 바로 *보안 요구사항*과 *오용 사례*이다. 본 가이드는 높은 수준의 보안 요구사항을 정의하는 것과 가장 많이 오용하는 시나리오를 해결하는 데 도움이 된다.

웹 개발팀은 조금이라도 보안에 이점을 줄 수 있는 사용자 측의 입력값 검증, 숨겨진 필드와 인터페이스 통제(예: 풀다운 메뉴와 라디오 버튼)와 같은 사용자 측 통제를 이해하는 것이 중요하다. 공격자는 애플리케이션 트래픽을 분석하고 조작된 요청을 전송하거나 모든 인터페이스를 우회하기 위해 사용자 측 웹 프록시(예: **OWASP WebScarab, Burp**) 또는 네트워크 패킷 캡처 도구(예: **WireShark**)와 같은 도구를 사용할 수 있다. 추가적으로 플래쉬, 자바 애플릿, 그리고 다른 사용자 측 오브젝트는 취약점을 찾기 위해 디컴파일되고 분석될 수 있다.

소프트웨어 보안 취약점은 다음을 포함하는 소프트웨어 개발 생명주기의 어떤 단계에서도 발견될 수 있다:

- 이전 보안 요구사항에 정의되지 않았을 때
- 논리적 에러를 가지고 있도록 개념적으로 설계했을 때
- 기술 취약점이 노출된 취약한 코딩 규칙을 사용했을 때
- 소프트웨어를 부적절하게 배치했을 때
- 관리 또는 업데이트 중 취약점이 발표될 때

더욱이, 소프트웨어 취약점은 소프트웨어 자체를 넘어서서 그 이상의 피해를 초래한다는 사실을 알아야 한다. 소프트웨어 및 취약점, 관련 인프라스트럭처의 성격에 따라서, 취약점 공격이 성공했을 때의 다음 중 하나 또는 모든 항목에 대해서 영향을 줄 수 있다:

- 소프트웨어 및 관련된 정보
- 관련 서버의 운영체제
- 데이터베이스
- 공유 환경에 있는 다른 애플리케이션
- 사용자 시스템
- 사용자와 상호작용하는 다른 소프트웨어

시큐어 코딩 규칙 체크리스트

입력값 검증:

- 신뢰 시스템의 모든 데이터 유효성을 검증 (예: 서버)
- 모든 데이터의 출처를 확인하고 신뢰/비신뢰로 구분하라. 출처를 신뢰하지 못하는 모든 데이터를 검증 (예: 데이터베이스, 파일 스트림 등)
- 애플리케이션의 입력값 검증 루틴은 중앙 집중화(단일화)
- 모든 입력값에 대해 **UTF-8** 과 같은 단일 문자셋(**character set**) 정의
- 데이터를 검증하기 전에 단일 문자셋으로 인코딩(*정형화*)
- 입력값 검증을 실패했을 때는 입력(처리) 거부
- 시스템이 **UTF-8** 확장 문자셋을 지원한다면, **UTF-8** 디코딩 후 검증
- 데이터를 처리하기 전, 모든 파라메터, **URL** 과 **HTTP** 헤더(예: 쿠키 명과 값)을 포함한 사용자 측에서 전송된 모든 데이터를 검증하라. 자바스크립트, 플래쉬, 다른 임베디드 코드에서 자동 생성한 **POST** 값도 포함하는 것을 잊어서는 안된다.
- 요청과 응답 헤더가 **ASCII** 문자만 포함하는 지 검증
- 리다이렉트된 데이터 검증(공격자는 리다이렉트되는 목표로 직접 악의적인 내용을 전송할 수 있다. 이를 통해, 리다이렉트 되기 전 수행되는 애플리케이션 로직과 입력값 검증을 우회할 수 있다.)
- 예상되는 데이터 형 검증
- 데이터 범위 검증
- 데이터 길이 검증
- 특수한 상황을 제외하고, 허용된 문자의 "화이트" 리스트에 맞춰 모든 입력값 검증
- 잠재적으로 *위험한 문자*가 반드시 입력되어야 한다면, 출력값 인코딩, 안전한 작업을 위한 **API**, 그리고 애플리케이션 전체에 대한 데이터가 활용되는 감사 기능과 같은 추가적인 통제기능 구현. 일반적으로 위험한 문자의 예는 다음과 같다:
  < > " ' % ( ) & + \ \' \"
- 표준 검증 루틴이 다음 입력값을 처리하지 못한다면, 개별적으로 명확하게 점검
  - 널 바이트 점검 (%00)
  - 개행 문자 점검 (**%0d, %0a, \r, \n**)
  - 상대경로(../ 또는 ..\) 변경 문자 (**UTF-8** 확장 문자셋 인코딩이 지원되는 경우 **%c0%ae%c0%ae/**와 같은 추가적인 변경이 가능하다. 이중 인코딩 또는 다른 형태의 난독화 공격을 해결하려면 *정형화(canonicalization)*를 활용하라)

출력 값 인코딩:

- 신뢰된 시스템의 모든 인코딩을 적용(예: 서버)
- 외부로 나가는 인코딩에 대해 표준적이고, 시험이 된 루틴 이용
- 애플리케이션의*신뢰범위(trust boundary)* 외부로 부터 클라이언트로 반환된 모든 데이터는 *상황에 맞는 출력 값 인코딩*. *HTML 엔터티 인코딩*이 예가 될 수 있지만, 모든 경우에 그런 것은 아니다.
- 사용하는 인터프리터에서 안전하다고 알려진 문자 이외의 *모든 문자를 인코딩*
- **SQL, XML,** 그리고 **LDAP** 에 쿼리하는 신뢰받지 못하는 데이터의 모든 출력 값을 상황에 맞게 *필터링*

- 운영체제 명령어로 전달되는 신뢰받지 못한 데이터의 모든 출력 값을 *필터링*

인증과 패스워드 관리:

- 의도적으로 공개하는 경우를 제외한 모든 페이지와 자원은 인증 실시
- 모든 인증 통제는 반드시 신뢰 시스템에서 수행(예: 서버)
- 특수한 경우를 제외하곤, 표준화되고 검증된 인증 서비스를 확립하고 이용
- 외부 인증 서비스를 호출하는 라이브러리를 포함한 모든 인증 통제는 중앙 집중화하여 사용하라
- 요청된 자원과 인증 로직을 구분하고 중앙 집중화된 인증 통제 정보를 주고받는 데는 리다이렉트를 사용하라
- 모든 인증 통제는 안전하게 인증 실패 처리
- 모든 관리자 기능과 계정 관리 기능은 최소한 주요한 인증 메커니즘 만큼 안전하게 관리
- 애플리케이션이 인증정보를 저장해야 하는 경우, 패스워드의 경우 암호학적으로 강한 단방향 해쉬값으로 저장되도록 하고, 패스워드와 키를 저장하는 테이블/파일은 해당 애플리케이션만 쓰기가 가능하도록 보장
- 패스워드 해쉬 연산은 반드시 신뢰되는 시스템에서 수행(예: 서버)
- 모든 데이터 입력이 완료되었을 때만 인증 데이터를 검증. 특히 *순차적 인증(sequential authentication)* 을 수행할 때 주의하라
- 인증 실패 메시지는 어떤 인증 정보가 틀렸는지를 나타내서는 안된다. 예를 들면, "잘못된 사용자명" 또는 "잘못된 패스워드"를 대신하여, 두 경우 모두에 대해 단지 "잘못된 사용자명/패스워드"라고 명시한다. 에러 응답은 이 두 경우에 대해 동일하게 표시함은 물론 소스코드도 동일해야 한다.
- 민감 정보 또는 기능과 관련된 외부 시스템에 연결시에도 인증을 활용
- 애플리케이션의 외부 서비스 접근을 위한 인증 정보는 암호화하여 신뢰받는 시스템의 보호된 영역에 저장(예: 서버). 소스 코드는 안전한 장소가 아니다.
- 인증 정보를 전송할 때에는 반드시 **HTTP POST** 요청을 사용
- 패스워드는 반드시 암호화된 통신 또는 암호화된 이메일과 같은 암호화된 데이터로 전송. 이메일 리셋과 같은 임시 패스워드는 예외가 될 수 있다
- 정책 또는 규제에 따른 패스워드 복잡성 요구사항을 적용. 인증 정보는 운영(배치) 환경에서 일반적 위협이 있는 공격에 충분히 대응할 수 있어야 함. (예: 알파벳 사용과 더불어 숫자 또는 특수문자를 요구)
- 정책 또는 규제에 따라 패스워드 길이 요구사항을 적용. 8 자리가 일반적으로 사용되지만, 16 자리가 더 나을 수 있으며, 패스워드 문구를 사용하는 것도 고려해보라.
- 패스워드 입력은 사용자 화면에서 드러나선 안 된다. (예: 웹 폼에서 사용되는 **input type "password"**)
- 사전에 정의된 로그인 실패 시도 후엔 계정 잠금을 적용(예: 일반적으로 5 번 시도를 허용한다). 계정은 인증에 대한 무차별 대입 공격(**brute force guessing**)을 대응할 수 있도록 계정에 대한 시간길이를 반드시 비활성화해야 하지만, 서비스 거부 공격(**denial-of-service**)이 가능할 정도로 너무 긴 시간동안 허용해서는 안됨.
- 패스워드 재설정과 변경 작업은 계정 생성과 인증과 동일한 수준의 통제가 요구된다.
- 패스워드 재설정에 사용되는 질문은 예측 불가능한 답변이 되어야 한다. (예: "좋아하는 책"은 좋은 예가 아니다. 왜냐하면, 가장 일반적인 답이 "성경"이기 때문이다.)

- 이메일을 기반으로 패스워드 재설정 기능을 사용할 경우, 사전에 등록된 주소로만 임시 링크 또는 임시 패스워드를 전송하라.
- 임시 패스워드와 임시 링크는 짧은 기간동안만 유효하도록 해야 함
- 다음 사용 시에는 임시 패스워드를 변경하도록 해야함
- 패스워드 재설정이 된 경우 사용자에게 알려라
- 패스워드 재사용을 금지하라
- 비밀번호 재사용으로 인해 발생 가능한 공격을 예방하기 위해 패스워드는 최소한 하루 이상 지난 후에 변경 가능해야 함
- 정책 또는 규제에 따라 패스워드 변경 주기를 적용. 주요 시스템은 보다 잦은 변경을 요구할 수 있다. 재설정 시간은 관리자가 통제해야 한다.
- 패스워드 폼 필드에 대한 필드값 자동 저장을 비활성화
- 사용자의 가장 최근 사용기록(성공적인 로그인 또는 실패한 로그인) 을 사용자가 다음 번 로그인했을 때 그 사용자에게 알려주도록 함
- 동일한 패스워드를 사용한 다중 사용 계정에 대한 공격을 인지할 수 있도록 모니터링하라. 이 공격 유형은 사용자 ID 를 수집했거나 추측 가능할 때 표준 잠금을 우회하기 위해 자주 사용된다
- 벤더가 제공한 모든 기본 패스워드와 사용자 ID 를 변경하거나 관련 계정을 비활성화
- 중요한 작업을 수행하기 전에는 사용자를 재인증
- 매우 민감하거나 매우 중요한 업무 계정에 대해선 *다중 인증(Multi-Factor Authentication)* 사용
- 인증을 위해 제 3 자 코드를 사용할 경우, 악의적인 코드에 영향을 받았는지 여부를 확인하기 위해 해당 코드를 주의깊게 점검하라

세션 관리:

- 서버 또는 프레임워크의 세션 관리 통제기능을 사용. 애플리케이션은 이들 세션 식별자만을 유효하다고 인지해야 함
- 세션 식별자 생성은 항상 신뢰된 시스템에서 이루어져야 한다 (예: 서버)
- 세션 관리 통제기능은 세션 식별자의 무작위성(random)을 보장할 수 있도록 엄격히 심사된 알고리즘을 사용해야 함
- 인증된 세션 식별자를 포함한 쿠키에 대해 도메인과 경로를 설정하여 사이트에 대한 적절히 제한
- 로그아웃 기능은 관련된 세션과 접속을 완벽하게 종료해야 함
- 로그아웃 기능은 인증에 의해 보호되는 모든 페이지에서 적용되어야 함
- 세션 비활성 타임아웃(session inactivity timeout)은 위험과 비즈니스 기능 요구사항의 균형에 기반하여 가능한 짧아야 설정. 대부분의 경우 3~5 시간정도를 초과해서는 안됨
- 장기적으로 로그인하는 기능을 허가하지 말고 세션이 활성화되어 있더라도 주기적으로 세션 종료. 특히 네트워크 접속이 많은 경우(rich network connections) 또는 주요한 시스템의 접속을 지원하는 애플리케이션의 경우에 적용해야 함. 부정적 효과를 최소화하려면 종료 시간은 비즈니스 요구사항을 따라야 하며, 사용자에겐 종료 이전에 충분한 알려야 함
- 세션이 로그인 이전에 생성된다면, 로그인 후에는 이전 세션을 종료하고 새로운 세션을 생성
- 재인증 할때엔 언제나 새로운 세션 식별자를 생성
- 동일한 사용자 ID 에 대해 동시(복수) 로그인을 허용하지 말 것

- URL, 에러 메시지, 로그에 세션 식별자를 노출하지 말 것. 세션 식별자는 오직 HTTP 쿠키 헤더에만 존재해야 한다. 예를 들면, 세션 식별자를 GET 인자로 전달하지 말 것
- 서버의 다른 사용자와 서버의 다른 접근 통제를 악용해 서버 측 세션 데이터에 불법적인 접근을 하지 못하도록 할 것
- 새로운 세션 식별자를 생성하고 주기적으로 지난 세션 식별자를 비활성화(이는 인증 식별자가 노출되었때 발생할 수 있는 세션 하이재킹 시나리오를 예방할 수 있음)
- 인증 중 과정 중 접속 보안이 HTTP 에서 HTTPS 로 변경되었땐 새로운 세션 식별자를 생성. 애플리케이션 내부에선 HTTP 에서 HTTPS 로 변경하는 것보다 지속적으로 HTTPS 활용할 것을 장려
- 계정 관리와 같은 민감한 서버 측 기능(업무)을 위해 세션별로 강력한 무작위 토큰 또는 파라메터를 이용해 표준 세션 관리를 보완. 이 방법은 *크로스 사이트 요청 위조(Cross Site Request Forgery)* 공격을 예방할 수 있음
- 매우 민감하거나 중요한 기능(업무)을 위해 세션별이 아닌 요청별로 강력한 무작위 토큰 또는 파라메터를 이용해 표준 세션 관리를 보완
- TLS 접속을 통해 전송되는 쿠키에 대해 "secure" 속성을 설정
- 쿠키 값을 읽거나 설정을 위해 사용자 측 스크립트를 애플리케이션에서 사용하지 않는다면, 쿠키에 HttpOnly 속성을 설정

접근 통제:

- 접근 권한 결정을 할 때엔 신뢰된 시스템 객체(예: 서버 측 세션 객체)만을 사용
- 접근 권한을 확인할 때에는 한 사이트 전체에 적용되는 한 개의 컴포넌트(single site-wide component)를 사용. 이는 외부 권한 서비스를 호출하는 라이브러리도 포함
- 접근 통제는 안전하게 실패를 처리하도록 해야 함
- 애플리케이션에서 보안 설정 정보에 접근할 수 없을 때에는 모든 접근을 거부
- 서버 측 스크립트에서 "includes"로 생성된 요청, AJAX 와 플래쉬 같은 리치 사용자 측(rich client-side) 기술로 생성된 요청을 포함하여 모든 요청에 대해 인증 통제를 적용
- 다른 애플리케이션 코드와 권한 로직을 구분
- 애플리케이션 직접 통제를 받지 않는 자원을 포함하여, 파일 또는 다른 자원에 대해 권한이 있는 사용자만이 접근할 수 있도록 제한
- 권한이 있는 사용자만 보호된 URL 에 접근할 수 있도록 제한
- 권한이 있는 사용자만 통제된 기능에 접근할 수 있도록 제한
- 권한이 있는 사용자에게만 직접 객체 참조 허용
- 권한이 있는 사용자에게만 서비스 접근 허용
- 권한이 있는 사용자에게만 애플리케이션 데이터 접근 허용
- 접근 통제에 사용되는 사용자, 데이터 속성, 그리고 정책 정보에 대한 접근 제한
- 권한이 있는 사용자에게만 보안 관련 설정 정보에 대한 접근 제한
- 접근 통제 정책의 서버 측 구현내용과 표현 계층(presentation layer)의 처리가 일치해야 함
- 사용자 측에 *상태 정보(state data)*가 저장되어야 하는 경우, 상태 조작을 탐지할 수 있도록 암호화와 서버 측에서 무결성 점검을 수행
- 애플리케이션 논리 흐름을 비즈니스 정책을 따르도록 적용

- 일정한 시간동안 한 명의 사용자와 장치가 수행할 수 있는 처리 건수를 제한. 처리 건수와 시간은 실제 비즈니스 요구보다는 많아야 하지만, 자동화 공격을 탐지할 수 있을 정도로 작아야 함
- "레퍼러(referer)" 헤더는 단지 참고 용도로만 사용. 레퍼러는 조작 가능하므로 단독 권한 점검으로 적합하지 못함
- 장시간의 인증 세션이 허용하면, 사용자가 권한을 가지고 있는지 주기적으로 사용자의 권한을 재확인하라. 권한을 가지고 있다면, 사용자를 로그아웃 시키고 재인증
- 계정 감사 기능을 구현하고 사용하지 않는 계정은 비활성화(예: 계정 패스워드의 종료일이 30 일 이상 지난 경우)
- 애플리케이션은 사용 권한이 중지되었을 때 계정을 비활성화하고, 세션 종료가 가능해야 함 (예: 역할, 고용 상태, 비즈니스 프로세스 등의 변화)
- 서비스 계정 또는 외부 시스템 접속을 지원하는 계정은 최소한의 권한을 가져야 함
- 접근이 적절하게 실행되고 통제될 수 있도록 애플리케이션의 비즈니스 역할, 데이터 유형, 접근 통제 표준과 프로세스를 문서화한 접근 통제 정책을 구축. 여기에 데이터와 시스템 자원 모두의 접근 요구사항도 포함

암호 규칙:

- 애플리케이션 사용자로의 비밀데이터를 보호하기 위해 사용되는 모든 암호화 기능은 신뢰 시스템에서 실행되어야 한다 (예: 서버)
- 인증되지 않은 접근으로 부터 마스터 키(master secrets) 보호
- 암호화 모듈은 안전하게 실패 처리
- 모든 무작위 숫자, 무작위 파일명, 무작위 GUID, 그리고 무작위 문자열이 추측 불가능한 값이어야 할 경우, 암호화 모듈의 검증된 난수 생성기를 사용하여 생성
- 애플리케이션에 사용되는 암호화 모듈은 FIPS 140-2 또는 동급의 표준을 준수 (http://csrc.nist.gov/groups/STM/cmvp/validation.html 참조)
- 암호화 키 관리 방법에 대한 정책과 프로세스를 확립하고 이용

에러 처리와 감사기록:

- 시스템 상세정보, 세션 식별자 또는 계정 정보를 포함한 민감 정보를 에러 메시지에 노출 금지
- 디버깅과 스택 추적 정보(stack trace information)를 현하지 않은 에러 핸들러 사용
- 일반 에러 메시지를 구현하고, 사용자 전용 에러 페이지 사용
- 애플리케이션은 서버에서 사용되는 에러처리를 사용하는 대신 애플리케이션 에러 처리
- 에러 조건이 발생한 경우 할당된 메모리를 알맞게 해제
- 보안 통제와 관련된 에러 처리 로직은 기본적으로 접근을 거부
- 모든 감사기록 통제는 신뢰된 시스템에서 수행(예: 서버)
- 감사기록 통제는 특정 보안 이벤트의 성공과 실패 모두를 지원해야 함
- 감사기록에는 중요한 *로그 이벤트 데이터(log event data)*를 포함하도록 보장
- 비신뢰 데이터를 포함한 로그 엔터리가 로그 열람 인터페이스 또는 소프트웨어에서 코드로써 실행되지 않도록 보장
- 권한이 있는 자에게만 감사기록 접근 허용
- 모든 감사기록을 운영할 때에는 마스터 루틴(master routine)을 활용

- 필요이상의 시스템 상세내용, 세션 식별자 또는 패스워드를 포함하여 민감한 정보를 감사기록에 저장해선 안 됨
- 감사기록 분석을 할 수 있는 메카니즘이 존재해야 함
- 모든 입력값 검증 실패를 감사 기록
- 모든 인증 시도, 특히 실패를 감사 기록
- 모든 접근 통제 실패를 감사 기록
- 의도되지 않은 상태 데이터 변경을 포함한 모든 눈에 띄는 부정시도를 감사 기록
- 모든 시스템 예외를 감사 기록
- 보안 설정 변화를 포함한 모든 관리자 활동을 감사 기록
- 모든 벡엔드 TLS 접속 실패를 감사 기록
- 암호화 모듈 실패를 감사 기록
- 감사 입력 무결성 확인을 위해 암호화 해쉬 함수 사용

데이터 보호:

- 기능, 데이터와 시스템 정보를 수행하는데 필요한 최소한의 권한을 부여하고 사용자를 제한
- 서버에 저장된 민감한 데이터의 캐쉬 또는 임시 파일에 대한 허가되지 않은 접근을 차단하고 더 이상 필요하지 않은 경우 즉각적으로 임시 작업 파일을 삭제
- 인증 확인 데이터와 같은 매우 민감한 저장 정보에 대해선 강력한 암호화를 적용(이는 서버 측에서도 동일하다). 항상 충분히 검증된 알고리즘를 사용. 추가적인 가이드는 "암호화 규칙" 참고
- 사용자가 다운로드 하는 서버 측 소스 코드를 보호
- 사용자 측에 패스워드, 접속 정보 또는 다른 민감한 정보가 일반 텍스트 또는 암호화 되지 않은 채로 저장해선 안됨. MS viewstate, 어도비 플래쉬 또는 컴파일된 코드와 같이 안전하지 않는 포맷으로 포함되어선 안됨
- 벡엔드 시스템 또는 다른 민감 정보를 노출할 수 있는 주석 정보를 사용자가 접근 가능한 코드에서 제거
- 공격자에게 유용한 정보를 노출할 수 있는 불필요한 애플리케이션과 시스템 문서를 제거
- HTTP GET 요청 파라메터에 민감한 정보를 제외
- 인증을 포함하여 민감한 정보가 포함될 수 있는 폼에 대해 자동 완성 기능 비활성
- 민감한 정보를 포함하는 페이지에서 사용자측 캐싱을 비활성화. HTTP 헤더 콘트롤 "Pragma: no-cache"와 결합하여 사용할 수 있는 Cache-Control: no-store 를 사용할 수 있다. Cache-Control: no-store 는 Pragma: no-cache 보다 비효율적이지만, HTTP/1.0 이후 버전과 호환된다.
- 애플리케이션은 민감한 데이터가 더 이상 필요하지 않은 경우 그 데이터를 삭제하는 기능을 포함해야 함 (예: 개인 정보 또는 특정 금융 데이터)
- 서버에 저장된 민감한 데이터에 대해 적절한 접근 통제 적용. 캐시 데이터, 임시 파일과 특정 시스템 사용자만 접근 가능한 데이터에서도 동일하게 적용

통신 보안:

- 민감한 정보 전송 시 데이터 암호화. 통신 보호를 위해 TLS 를 사용해야 하며, 민감한 파일 또는 HTTP 기반 접속이 아닌 경우에도 개별 암호화를 통해 보호
- TLS 인증서는 유효한 것을 사용하고, 올바른 도메인 명과 기한이 만료되지 않은 것 사용. 또한 필요한 경우에는 중개 인증서가 설치되어야 함
- TLS 접속이 실패한 경우 안전하지 않은 접속으로 다시 이동해선 안됨

- 인증된 접근이 요구되는 모든 콘텐츠와 모든 민감한 정보에 대해 TLS 접속을 활용
- 민감한 정보 또는 기능을 포함한 외부 시스템에 접근할 때 TLS 를 활용
- 올바르게 설정된 한 개의 표준 TLS 를 구현
- 모든 접속에 대해 문자 인코딩을 정의
- 외부 사이트로 연결할 때 HTTP referer 에 민감한 정보를 포함한 파라메이터가 전달되지 않도록 구현

시스템 설정:

- 서버, 프레임워크 그리고 컴포넌트가 검증된 최신 버전으로 운영
- 서버, 프레임워크 그리고 컴포넌트가 사용 중인 버전에 대해 발표된 모든 패치를 적용
- 디렉토리 리스팅을 비활성화
- 웹 서버, 프로세스 그리고 서비스 계정은 최소한의 권한으로 운영
- 예외가 발생한 경우 안전하게 예외처리
- 불필요한 기능과 파일을 제거
- 개발 이후 제품(완성품)에 필요하지 않은 테스트 코드와 기능을 제거
- robot.txt 파일을 통해 디렉토리 구조가 노출되지 않도록 하라. 공개 인덱싱을 허용하고 싶지 않은 디렉토리는 독립된 부모 디렉토리에 위치시킨 후, 개별 디렉터별로 거부하는 것보다 전체 부모 디렉토리에 대해 robot.txt 로 "Disallow"하라.
- 애플리케이션에서 어떤 HTTP 메소드(Get 또는 Post)를 지원할 지와 애플리케이션의 다른 페이지에서 이를 어떻게 다르게 다룰지 정의
- WebDAV 확장과 같은 불필요한 HTTP 메소드를 비활성화. 파일 처리를 지원하는 확장 HTTP 메소드가 필요한 경우, 잘 검증된 인증 메카니즘을 활용
- 웹 서버가 HTTP 1.0 과 1.1 를 함께 사용한다면, 둘 다 동일한 방식으로 설정되었거나 둘 사이에 존재할 수 있는 차이점을 반드시 이해해야 함 (예: 확장 HTTP 메소드의 처리)
- HTTP 응답 헤더에서 OS, 웹 서버 버전 그리고 애플리케이션 프레임워크와 관련된 불필요한 정보를 제거
- 애플리케이션에 저장된 보안 설정은 감사(audit)에 활용할 수 있도록 사람이 읽을 수 있는 출력물을 제공해야 함
- 자산 관리를 수행한 후 시스템 컴포넌트와 소프트웨어를 등록
- 운영 네트워크와 개발 환경을 격리하고 권한이 있는 개발 그룹과 테스트 그룹에게만 접근 권한을 부여. 개발 환경은 운영 환경보다 보안에 취약하게 설정되는 경우가 종종 있으며, 공격자는 이 차이점을 공유된 취약점 찾거나 공격의 수단으로 악용할 수 있다
- 개발 환경과 운영 환경 모두 소스 코드의 변화를 기록하고 관리할 수 있도록 소프트웨어 변경 통제 시스템을 적용

데이터베이스 보안:

- 반드시 *사전 준비된 쿼리(prepared queries)* 를 사용
- 입력값 검증과 출력 인코딩을 활용하고 메타 문자를 확실히 처리. 만약 이 작업이 실패한다면, 데이터베이스 명령어를 실행시켜선 안됨
- 반드시 변수가 제대로 입력되었는지 검증
- 데이터베이스에 접속했을 때 애플리케이션은 가장 낮은 수준의 권한을 사용

- 데이터베이스 접속을 위해 안전한 인증정보를 사용
- 접속 정보는 애플리케이션 내에 하드코딩 되어선 안됨. 접속 정보는 신뢰된 시스템의 별도 설정 파일로 저장 및 암호화
- 데이터에 직접 접근하지 못하도록 저장 프로시저(stored procedure)를 사용하고, 데이터베이스에 존재하는 기본 테이블에 대한 권한 제거
- 데이터베이스 접속은 가능한 빨리 종료
- 기본으로 제공되는 데이터베이스 관리 계정을 제거하거나 패스워드를 변경. 강력한 패스워드/비밀문구를 활용하거나 다중 인증을 구현
- 불필요한 모든 데이터베이스 기능을 비활성화(예: 불필요한 저장 프로시저 또는 서비스, 유틸리티 패키지, 필요한 최소한의 기능과 옵션만을 설치하라(공격 노출 범위 축소))
- 불필요한 벤더 기본 콘텐츠를 제거(예: 샘플 스키마)
- 비즈니스 요구사항에 필요하지 않은 모든 기본 계정을 비활성화
- 애플리케이션은 데이터베이스에 접속할 때 개별 기능(권한)에 대해 다른 인증정보를 사용 (예: 사용자, 읽기 전용 사용자, 방문자, 관리자)

파일 관리:

- 사용자가 입력한 데이터를 직접적으로 동적 include 함수로 전달하지 않도록 구현
- 파일 업로드 이전에 인증을 요구
- 비즈니스 목적에 필요한 파일 유형만 업로드 허용
- 업로드된 파일은 파일 헤더를 점검하여 파일 유형을 검증(파일 확장자만을 이용한 검증은 충분하지 않음)
- 파일을 애플리케이션과 동일한 웹 환경(디렉토리)에 저장해선 안됨. 파일은 또한 콘텐츠 서버나 데이터베이스에 저장되어서도 안됨
- 웹 서버에서 실행될 수 있는 파일은 업로드 되지 못하도록 하거나 업로드를 제한
- 파일 업로드 디렉토리의 실행 권한을 비활성화
- UNIX 에선 관련 경로 또는 chroot 환경을 이용한 논리 드라이브를 파일 저장 디렉토리로 마운팅하여 안전한 파일 업로드가 가능하도록 설정
- 존재하는 파일을 참조할 때는 허용된 파일 이름과 유형의 화이트 목록을 사용. 전달된 파라메터의 값을 확인하고 유효한 값이 아닐 경우, 요청을 거부하거나 요청된 값 대신 하드코딩된 기본 파일 값을 대신 사용하도록 함
- 사용자가 제공한 데이터를 동적 리다이렉트로 전달해선 안됨. 허용을 반드시 해야할 경우, 리다이텍트는 유효한 상대 경로 URL 만 허용
- 디렉토리 또는 파일 경로를 전달하지 않고, 사전에 정의된 경로 목록과 매핑된 인덱스 값을 사용
- 사용자 측에 절대 파일 경로를 결코 전송해선 안됨
- 애플리케이션 파일과 자원은 읽기 전용으로 설정
- 업로드된 파일에 대해 바이러스와 악성코드 여부를 점검

메모리 관리:

- 신뢰하지 못하는 데이터에 대해 입력값과 출력값을 통제
- 버퍼의 크기가 정의한 것보다 큰 지를 이중으로 점검

- strncpy()와 같은 특정 크기의 바이트 복사를 허용하는 함수를 사용할 때, 대상 버퍼 크기가 원본 버퍼 크기와 동일한지를 점검(종료 문자인 NULL 로 끝나지 않을 수 있다.)
- 반복문에서 함수를 호출했을 때 버퍼의 경계를 점검하고 과거 할당된 공간을 덮어쓰는 위험이 없는 지 확인
- 모든 입력 문자는 복사 또는 연결(concatenation) 함수로 전달되기 전에 적절한 길이로 분할
- 명확하게 자원을 종료 처리. 가비지 컬렉션(garbage collection)에 의존해선 안됨 (예: 접속 오브젝트, 파일 핸들러 등)
- 가능하다면 실행되지 않는(non-executable) 스택을 사용
- 잘 알려진 취약한 함수의 사용 자제 (예: printf, strcat, strcpy 등)
- 함수의 종료 또는 모든 exit 포인트에서 할당된 메모리를 정확하게 반환

일반 코딩 규칙:

- 일반 작업 시 관리되지 않은 새로운 코드를 사용하는 것보다 테스트되고 증명된 관리 코드를 사용
- 운영체제 작업을 처리 시 특정 작업을 위해 내장된 API 를 사용. 애플리케이션이 직접 운영체제의 명령어를 실행해선 안됨. 특히, 명령어 쉘이 포함된 애플리케이션의 사용을 통해 실행되도록 해선 안됨
- 해석된 코드, 라이브러리, 실행 파일 그리고 실행 파일에 대한 검증을 위해 체크섬 또는 해쉬를 사용
- 다수의 동시 요청을 예방하기 위해 잠금(locking)을 활용하거나 경쟁 상태(race condition)를 예방하기 위해 동기화 메카니즘을 사용
- 공유 변수와 자원에 대해 부적절한 동시 접근이 발생하지 않도록 보호
- 모든 변수와 다른 데이터 저장 요소는 선언 또는 최초 사용 전에 명확하게 초기화
- 애플리케이션이 상위 권한을 반드시 사용해야 하는 경우, 권한 상승은 가능한 마지막에 수행하고 최대한 빨리 원상 복귀
- 프로그래밍 언어의 기본 숫자 표현 방식과 숫자 계산 방법을 이해하여 계산 에러가 발생하지 않도록 처리. 바이트 크기의 불일치, 정밀성, signed/unsigned 구별, 버림(truncation), 얼마나 큰 수와 작은 수를 프로그래밍 언어에서 다룰 수 있는 지와 숫자 유형(무효수치(not-a-number) 계산) 간의 환산(conversion)과 변환(casting)을 주의깊게 검토
- 사용자가 전송한 데이터를 동적 실행 함수에 절대 전달해선 안됨
- 사용자가 새로운 코드를 생성하거나 기존의 코드를 수정하도록 해선 안됨
- 모든 2 차 애플리케이션, 제 3 자 코드와 라이브러리에 대해 비즈니스 필요성을 검토하고, 새로운 취약점을 초래할 수 있는 기능의 안전성 확인
- 안전하게 업데이트하는 기능을 구현. 애플리케이션이 자동 업데이트를 활용한다면, 소스 코드에 대해 암호 시그니처를 사용하고 다운로드 프로그램에서 시그니처를 검증. 호스트 서버에서 코드를 전송할 땐 암호화된 채널을 사용

부록 A:

외부 참고 문헌:

1. 인용한 참고문헌
   SANS 와 TippingPoint " 탑 사이버 보안 위험 "
   http://www.sans.org/top-cyber-security-risks/

- 웹 애플리케이션 보안 컨소시엄
  http://www.webappsec.org/

- Common Weakness Enumeration (CWE)
  http://cwe.mitre.org/

- 홈랜드 시큐리티의 소프트웨어 보증 프로그램 부서
  안전한 포탈 구축하기
  https://buildsecurityin.us-cert.gov/daisy/bsi/home.html

- CERT 시큐어 코딩
  http://www.cert.org/secure-coding/

- MSDN 보안 개발 센터
  http://msdn.microsoft.com/en-us/security/default.aspx

- SQL 인젝션 참고 목록
  http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/

- Cross Site Scripting (XSS) 참고 목록
  http://ha.ckers.org/xss.html

보안 권고 사이트:

인프라스트럭처와 프레임워크를 위협하는 알려진 보안 취약점을 확인할 수 있는 유용한 자원

Secunia Citrix 취약점 목록:
- http://secunia.com/advisories/search/?search=citrix

Security Focus 취약점 검색:
- http://www.securityfocus.com/vulnerabilities

오픈 소스 취약점 데이터베이스 (OSVDB):
- http://osvdb.org/search/web_vuln_search

CVE(Common Vulnerability Enumeration):
- http://www.cve.mitre.org/

부록 B: 용어집

오용 사례(Abuse Case): 소프트웨어의 의도적 오용과 비의도적 오용을 말한다. 오용 케이스는 시스템 설계에서 가정(전제)에 대한 대한 위협요소이다.

접근 통제(Access Control): 시스템 자원에 접근하려는 사용자 또는 자원에 대해 허용하고 거부하는 일련의 통제. 일반적으로 역할내 계층적인 역할과 개인의 권한, 그리고 시스템 상호간의 연관성을 기반으로 한다.

인증(Authentication): 소프트웨어와 상호 작용하는 사용자 또는 다른 엔터티의 신원을 확인하는 일련의 통제

가용성(Availability): 시스템의 접근성과 사용성의 척도

정규화(Canonicalize): 간단한 단일 형식을 이용해 다양한 인코딩과 데이터 표현을 축소하는 것

통신 보안(Communication Security): 소프트웨어가 안전하게 정보를 송수신하도록 하는 일련의 통제

비밀성(Confidentiality): 정보가 권한이 있는 자에게만 공개되는 것을 보장하는 것

상황에 맞는 출력 값 인코딩(Contextual Output Encoding): 애플리케이션에서 어떻게 데이터를 활용할지를 고려하여 결과를 인코딩하는 것. 인코딩 방법은 사용된 결과 데이터의 사용 목적에 따라서 다양하다. 클라이언트 측의 응답에 데이터가 포함된다면, 다음을 포함하는 시나리오를 설명해야 한다: HTML 문서의 본문, HTML 속성, 자바스크립트, CSS, URL. 또한 SQL 쿼리, XML, LAP 와 같은 다른 경우에 대해서도 설명해야 한다.

크로스 사이트 요청 위조(Cross Site Request Forgery): 외부 웹사이트 또는 애플리케이션이 사용자에게 의도하지 않은 요청을 유효한 세션을 맺고 있는 다른 애플리케이션을 보내는 것이다. 애플리케이션은 이해가능하거나 예측 가능한 URL 과 파라메터를 사용할 때 취약하다. 또한 취약한 애플리케이션에 대해 브라우저가 모든 필요한 세션 정보를 자동적으로 한꺼번에 요청할 때 발생 가능하다. (관련 취약점이 매우 일반적이지만 사람들의 이해가 부족하기 때문에 공격유형으론 유일하게 본 문서에서 상세한 설명을 한다)

암호화 규칙(Cryptographic Practices): 애플리케이션의 암호연산이 안전하게 처리됨을 보장하는 일련의 통제

데이터 보호(Data Protection): 소프트웨어가 안전하게 정보를 저장하도록 처리하는 일련의 통제

데이터 베이스 보안(Database Security): 소프트웨어가 데이터베이스와 안전한 방법으로 상호작용하고 데이터베이스가 안전하게 설정되었음을 보장하는 일련의 통제

에러 처리와 감사기록(Error Handling and Logging): 애플리케이션이 에러를 안전하게 처리하고 적절하게 이벤트를 감사기록하는 것을 보장하는 일련의 규칙

악용(Exploit): 취약점을 이용해 권한 또는 정보를 취하는 행위. 일반적으로 취약점을 찾아내어 소프트웨어의 보안 통제를 취약하게 하는 의도적인 행위이다.

파일 관리(File Management): 코드와 다른 파일 시스템 파일간의 상호작용을 다루는 일련의 통제.

일반 코딩 규칙(General Coding Practices): 특정 카테고리에 속하기 힘든 코딩 규칙을 다루는 일련의 통제.

위험 문자(Hazardous Character): 애플리케이션에서 의도한 의미를 가지거나 관련 시스템에서 원래의 의미를 벗어나 특수한 의미로 해석되는 문자 또는 인코딩된 문자 표현. 이들 문자는 다음과 같은 결과를 초래할 수 있다:

- 기존의 코드 또는 구문의 구조를 변경
- 의도되지 않은 새로운 코드를 삽입
- 경로 변경
- 프로그램 함수 또는 루틴이 의도하지 않은 결과 산출
- 에러 조건 발생
- 하위 애플리케이션 또는 시스템에 앞선 결과를 초래

HTML 개체 인코딩(HTML Entity Encode): ASCII 문자를 동일한 의미의 HTML 엔터티로 변경하는 과정. 예를 들어, 인코딩은 문자 "<"는 동일한 의미의 HTML "<"로 변경한다. HTML 엔터티는 대부분의 인터프리터(특히, 웹 브라우저)에서 특수한 의미를 가지지 않으며, 특정 사용자 측 공격을 완화할 수 있다.

피해(Impact): 의도하지 않은 사건의 발생으로 인해 비즈니스에 초래될 수 있는 부정적 영향의 정도; 취약점을 악용함으로써 발생할 수 있는 결과는 무엇인가?

입력값 검증(Input Validation): 모든 입력 데이터 속성이 애플리케이션에서 의도한 자료형, 길이, 범위, 수용 가능한 문자셋과 일치하고 알려진 위험 문자가 포함되지 않았는지를 확인하는 통제

무결성(Integrity): 정보의 정확성, 완벽성, 유효성, 그리고 권한이 없는 행위로 인해 변경되지 않았다는 것을 보장하는 것.

로그 이벤트 데이터: 로그 이벤트 데이터는 다음을 포함해야 한다:

1. 신뢰하는 시스템 컴포넌트의 타임 스탬프
2. 각 이벤트의 심각도
3. 다른 로그 엔트리와 섞여있는 경우 관련 이벤트에 대한 태깅
4. 해당 이벤트를 초래한 계정 또는 사용자의 신원(식별자)
5. 요청과 관련된 근원 IP 주소
6. 이벤트 결과(성공 또는 실패)
7. 이벤트에 대한 설명

메모리 관리(Memory Management): 메모리와 버퍼 사용을 관리하는 일련의 통제.

완화(Mitigate): 취약점의 심각성을 감소시키기 위한 절차. 이는 취약점 제거, 취약점을 악용하기 어렵도록 조치, 공격이 성공했을 때의 부정적 피해 감소 등을 포함한다.

다중 인증(Multi-Factor Authentication): 사용자가 다수의 다양한 형태의 인증정보를 사용하는 인증 과정. 일반적으로 사용자가 소유한 것(예: 스마트카드), 사용자가 아는 것(예: 개인식별번호), 자신 그 자체(예: 생체정보)를 기반으로 한다.

출력 값 인코딩(Output Encoding): 애플리케이션의 출력 데이터로 인한 위험을 제거하기 위해 인코딩을 사용하도록 하는 일련의 통제.

매개변수 쿼리(Parameterized Queries (prepared statements)): 플레이스홀더(placeholder)를 사용하여 쿼리와 데이터가 구별되도록 한다. 쿼리 구조는 플레이스홀더로 정의된 후 SQL 구문은 데이터베이스에 전송되어 준비된 후, prepared statmenet 가 파라메터와 결합한다. 이는 파라메터 값이 SQL 문자열이 아닌 컴파일된 쿼리와 결합하기 때문에 원래의 쿼리가 변경되는 것을 예방한다.

데이터 검열(Sanitize Data): 데이터 제거, 변경, 인코딩, 문자제외를 통해 잠재적으로 위험한 데이터를 안전하게 변경하는 과정.

보안 통제(Security Controls): 잠재적 취약점을 완화하고 소프트웨어가 의도한 대로 동작하도록 보장하는 행위.

보안 요구사항(Security Requirements): 소프트웨어가 안전한 방법으로 개발되고 배치되도록 돕는 일련의 설계 및 기능 요구사항의 집합

순차적 인증(Sequential Authentication): 인증 데이터가 단일 페이지에서 한번에 처리되는 것이 아니라 인증 성공 페이지까지 순차적으로 요청하는 것

세션 관리(Session Management): 웹 애플리케이션이 HTTP 세션을 안전하게 처리하도록 하는 일련의 통제

상태 데이터(State Data): 애플리케이션 또는 서버에서 사용하는 데이터 또는 파라메터로, 지속적인 접속을 유지하거나 다중 요청 과정 또는 거래를 추적하기 위해 사용한다.

시스템(System): 운영체제, 웹 서버, 애플리케이션 프레임워크와 관련 인프라 스트럭처를 포괄하는 일반 용어

시스템 설정(System Configuration): 소프트웨어의 인프라스트럭처 컴포넌트가 안전하게 배치될 수 있도록 돕는 일련의 통제

위협 요소(Threat Agent): 시스템에 부정적 영향을 미칠 수 있는 임의의 개체. 시스템의 보안 통제를 침해할려는 악의적인 사용자가 일반적 위협 요소이지만, 화재 또는 홍수 등과 같은 물리적 위협과 실수로 인한 시스템 오작동도 이에 포함된다.

신뢰 범위(Trust Boundaries): 일반적으로 신뢰 범위는 직접적인 통제가 가능한 시스템 컴포넌트로 구성된다. 직접적인 통제가 불가능한 외부 시스템(모든 사용자와 협력자가 운영하는 시스템을 포함)으로부터의 모든 접속과 데이터는 비신뢰 구간에 속하며, 내부로 유입 전(다른 시스팀과 상호 작용하기 전)에 검증해야 한다.

취약점(Vulnerability): 시스템을 공격 또는 손상되기 쉽도록 하는 약점.