스마트 컨트랙트의 보안 내재화를 위한 체크리스트

스마트 컨트랙트의 보안 내재화를 위한 체크리스트

Institute : 고려대학교 블록체인 연구센터

Version : 2.0

<제목 차례>

<표 차례>

1. 목적

스마트 컨트랙트는 블록체인 기술의 중요 혁신 기술의 하나로, 계약 조건이 코드로 명시되어 자동으로 실행되는 디지털 계약을 의미한다. 스마트 컨트랙트의 개념은 1994년 닉 사보(Nick Szabo)에 의해 제안 되었으며, 특정 조건이 충족되면 계약이 자동으로 이행되는 방식으로 신뢰성과 보안을 크게 향상 시킬수 있다고 제안하였다. 스마트 컨트랙트의 가장 큰 장점은 인간의 개입 없이도 안전하고 신뢰할수 있는 거래를 가능하게 하여, 금융, 법률, 공급망 관리 등 인간의 사회에서 매우 중요한 분야에서 활용될수 있는 장점을 가지고 있다는 점이다.

스마트 컨트랙트는 주로 이더리움 (Ethereum)과 같은 블록체인 플랫폼에서 실행된다. 이더리움은 Turing Complete 가상 머신인 이더리움 가상머신 (Ethererum Virtual Machine)을 통해 스마트 컨트랙트를 실행할수 있는 환경을 제공하며, 이를 통해 다양한 산업 분야에서 스마트 컨트랙트가 실질적으로 구현되고 활용할수 있다. 이러한 기술의 발전은 스마트 컨트랙트가 기존의 중앙 집중식의 단점을 대체하거나 보완할 수 있고 또한 혁신적인 솔루션을 제공한다.

스마트 컨트랙트를 통해 여러 경제적인 활동을 함에 따라 악의적인 공격자에 의해 공격 대상이 될 수 있고, 여러 보안 위협에 노출될 수 있다. 스마트 컨트랙트는 자동화된 코드로 동작하기 때문에, 코드의 오류나 설계상의 결함이 있을 경우 예상치 못한 결과를 초래할 수 있다. 예를 들어, 스마트 컨트랙트의 잘못된 구현으로 인해 재진입 공격과 같은 취약점에 노출될 수 있으며, 이는 공격자가 스마트 컨트랙트의 자산을 무한정으로 인출하는 등의 심각한 보안 사고로 이어질 수 있다. 스마트 컨트랙트는 개발 단계에서 지속적인 코드 검토와 테스트가 필수이며, 만일 어느 한 과정이라도 생략하게 되면, 다양한 버그와 취약점을 가질 수 있다.

이러한 취약점은 스마트 컨트랙트의 실행과 보안에 직접적인 영향을 끼친다. 따라서 소프트웨어 관점이나 네트워크 관점과 같은 다양한 관점에서 스마트컨트랙트의 보안 내재화를 할수 있는 체계적인 방법이 중요하다.

따라서 본 문서는 스마트 컨트랙트의 보다 효율적인 보안 내재화를 위하여 개발 단계에서

필요한 보안 요구사항을 도출하여, 스마트 컨트랙트의 개발 초기 단계인 요구사항 분석부터 설계, 구현까지의 보안요구사항을 제공하려고 한다. 이에 따라 스마트 컨트랙트를 위한 위협 완화 프로세스를 기반으로 정략적 보안약점 점수 도출 및 산정 후 취약점 점수를 바탕으로 개발자들이 요구사항 분석서부터 구현단계 과정에 있어서 보안내재화의 효율적으로 적용할 수 있는 보안요구사항 체크리스트를 도출 및 제시한다.

2. 용어 정의 및 약어

1. **위협 (Threat)**: 의도적이거나 그렇지 않거나 취약점을 악용하여 자산, 정보에 손상, 피해, 파괴 등을 할 수 있는 모든 것

2. **위험(Risk)** : 취약점이 악용된 위협으로부터 발생한 정량적인 손상, 피해, 파괴 정도

3. **보안 약점(Weakness)** : 소프트웨어 내 포함되는 bug, flaw, mistake, error 등 모든 비정상 코드

4. **취약점(Vulnerability)** : 위협으로 악용되어 자산에 무단으로 접근할 수 있는 프로그램의 약점

5. **공격(Attack)** : 자산 및 시스템에 접근하거나 손상을 입힐 수 있는 시도

6. **CWSS(Common Weakness Scoring System)** : 소프트웨어 내에 포함된 버그(bug), 결함(Flaw), 실수(mistake), 오류(error) 등 모든 비정상 코드가 악용되어 자산에 무단으로 접근할 수 있는 위협으로 이어질 가능성을 평가하고 우선순위를 매기는 방법론.

7. **체크리스트 (Checklist)**: 체크리스트는 소프트웨어 개발 과정에서 품질, 보안, 성능, 요구사항 준수 등을 체계적으로 검토하고 확인하기 위해 사용하는 항목들의 목록으로, 이를 통해 개발자가 소프트웨어가 기대되는 기준을 충족하는지 확인하고, 잠재적인 문제를 사전에 발견하여 개선할 수 있도록 돕는 도구.

8. **보안요구사항(Security Requirements)** : 소프트웨어 시스템이 안전하게 동작하고 자산을 보호하기 위해 충족해야 하는 조건이나 기준을 의미하며, 이는 데이터 무결성, 기밀성,

접근 제어, 인증, 감사 추적 등과 같은 보안 측면을 포함하여, 시스템이 외부 및 내부의 위협으로부터 보호될 수 있도록 설계와 구현에 반영되어야 하는 요구사항.

3. 스마트 컨트랙트의 보안위협 실제 사례 조사

1. The DAO(Decentralized Autonomous Organization) 공격 (2016)

The DAO(Decentralized Autonomous Organization)는 2016년에 설립된 탈중앙화 자율 회사로, 이더리움 블록체인 상에서 스마트 컨트랙트를 활용해 자율적으로 운영되는 투자 펀드를 운영하는 회사이다. The DAO는 투자자들이 투표를 통해 자금을 투자할 프로젝트를 선택할 수 있도록 설계되었으며, 당시 큰 혁신적인 모델로 큰 관심을 받았다. 프로젝트는 약 1.5억 달러의 자금을 모으는 데 성공했지만, 스마트 컨트랙트 코드에 잠재적인 보안 약점이 존재했다.

공격자는 이 취약점을 이용하여 이더리움을 전송하는 함수가 호출될 때마다 외부 컨트랙트를 통해 원래 컨트랙트로 재진입(Reentrancy)할 수 있도록 공격을 진행하였다. 결과적으로 이더리움 잔액이 갱신되기 전에 동일한 자금을 여러번 인출 할 수 있었고, 그 과정을 반복함으로써 공격자는 한 번의 트랜잭션으로 다수의 인출을 실행해 3.6백만 이더리움을 탈취하였다.

2. Parity Wallet 해킹 공격 (2017)

Parity Wallet은 이더리움 기반의 다중 서명 지갑(multi-signature wallet)으로, 여러 사용자가 서명을 해야만 자금을 인출할 수 있도록 설계되었다. 이 지갑은 특히 ICO(Initial Coin Offering) 및 블록체인 프로젝트에서 자금을 관리하는 데 널리 사용되었다. 하지만 2017년에 두 차례의 공격이 발생하여 상당한 자금 손실을 초래하였다.

첫 번째 공격에서는 공격자는 스마트 컨트랙트의 초기화 되지 않은 함수를 악용하여 지갑의 소유권을 탈취하였다. Parity Wallet의 스마트 컨트랙트에는 초기화 코드가 남아 있었는데 이 코드를 통해 누구나 관리자 권한을 획득할수 있게 되었다. 공격자는 이 취약점을 통해 지갑의 소유권을 가로 채고 그 결과, 약 15만 이더리움을 도난 하였다.

두 번째 공격에서는, Parity Wallet의 사용자가 실수로 Self- Destruct 함수를 실행하여 발생한 것으로, 지갑에 있는 513,774 이더리움이 동결 되었다. Self Destruct 함수는 스마트 컨트랙트를 삭제 하는 기능으로, 이를 실행하면 해당 컨트랙트와 연관된 모든 자산이 접근 불가

능 해진다. 이 사고는 사용자 계정이 스마트 컨트랙트를 관리할 수 있도록 허용한 설계 결함에서 기안했으며 이로 인해 막대한 자산이 영구적으로 잠기게 되었다.

3. Bancor 해킹 공격 (2018)

Bancor는 암호화폐의 자동화된 변환을 가능하게 하는 탈중앙화 유동성 네트워크이다. Bancor의 스마트 컨트랙트는 사용자들이 서로 다른 암호화폐를 교환할수 있도록 설계 되었으며, 중앙화된 거래소의 필요성을 제거 하는 플랫폼이다.

2018년 7월, Bancor의 스마트 컨트랙트에서 발생한 취약점을 이용해 약 1,300만 달러 상당의 암호화폐가 도난당하였다. 이 공격은 스마트 컨트랙트의 업데이트 기능을 악용해 이루어졌다. 스마트 컨트랙트는 코드가 블록체인에 배포되면 변경이 불가능하다는 특성이 있지만, Bancor는 일부 컨트랙트를 업데이트할 수 있도록 설계하였는데, 이 기능이 오히려 공격의 표적이 되었다.

공격자는 이 업데이트 기능을 통해 스마트 컨트랙트를 악의적으로 재설정하고, 이를 이용해 자금을 인출하였다. 이 사건은 스마트 컨트랙트의 업데이트 기능을 통해 보안 약점이 발생 할 수 있고, 공격이 가능 할 수 있다는 것을 보여주었다.

4. SpankChain 해킹 (2018)

2018년 10월, SpankChain의 스마트 컨트랙트가 재진입 공격에 노출되어 약 38,000달러 상당의 암호화폐가 도난당했다. 이 공격은 The DAO 해킹 사건과 유사한 방식으로, 스마트 컨트랙트가 외부 컨트랙트를 호출하는 과정에서 발생하였다.

공격자는 스마트 컨트랙트가 자금을 인출하는 함수를 호출할 때, 해당 함수를 재귀적으로 호출해 스마트 컨트랙트의 상태가 업데이트되기 전에 자금을 반복적으로 인출할 수 있었다. 이를 통해 공격자는 적은 양의 자금을 여러 번 인출하여 상당한 금액을 탈취하였다.

5. dForce 해킹 (2020)

2020년 4월, dForce의 imBTC 스마트 컨트랙트에서 발생한 재진입 공격으로 인해 약 2,500만 달러의 자금이 탈취되었다. 공격자는 스마트 컨트랙트가 다른 컨트랙트를 호출할 때의 취약점을 이용해 자금을 인출하기 전에 재진입하여 같은 자금을 여러 번 인출할 수 있었다.

공격은 두 단계로 이루어졌는데. 첫 번째 단계에서는 플래시 론(Flash Loan)을 통해 대규모 자금을 대출받아 dForce의 스마트 컨트랙트에 유입시켰고, 두 번째 단계에서는 재진입 공격을 통해 자금을 반복적으로 인출하여 막대한 이익을 취하였다.

6. Etherscan 해킹 (2018)

Etherscan은 이더리움 블록체인 탐색기로, 사용자가 이더리움 블록체인에서 트랜잭션, 주소, 스마트 컨트랙트 등을 검색하고 확인할 수 있는 도구이다. 2018년, Etherscan 웹사이트가 XSS 공격을 당하였다.

공격자는 Etherscan의 웹 페이지에 악성 스크립트를 삽입하여, 해당 페이지를 방문하는 사용자의 브라우저에서 악성 코드를 실행시켰다. 이 악성 코드는 사용자의 트랜잭션 데이터를 조작하거나 민감한 정보를 탈취하는 데 사용될 수 있었다.

7. Yearn Finance 해킹

Yearn Finance는 자동화된 수익 최적화 전략을 제공하는 DeFi 플랫폼이다. 2021년, Yearn Finance의 DAI(다이) 금고(vault)가 플래시 론 공격을 당하였다.

공격자는 플래시 론을 이용해 Cream Finance와 Alpha Homora 간의 통합에서 발생한 취약점을 공격하였다. 이 공격은 두 플랫폼 간의 상호작용에서 발생하는 논리적 결함을 이용한 것으로, 약 3,700만 달러의 자산이 탈취되는 사례를 발생하였다.

8. BadgerDAO 해킹 (2021)

BadgerDAO는 DeFi 프로토콜로, 비트코인을 사용하여 이익을 창출할 수 있는 다양한 금융 서비스를 제공하는 플랫폼이다. 2021년, BadgerDAO의 프론트엔드가 해킹 공격을 받았다.

공격자는 BadgerDAO의 웹사이트 프론트엔드에 악성 스크립트를 삽입하였다. 이 스크립트는 사용자가 트랜잭션을 승인할 때 공격자의 주소로 자산이 전송되도록 조작하였고. 이로 인해 약 1억 2,000만 달러 이상의 암호화폐가 도난당하는 사건이 발생하였다.

9. Pickle Finance 해킹

Pickle Finance는 탈중앙화된 금융 플랫폼으로, 다양한 수익 전략을 제공하며 사용자의 자산을 관리하는 수익성 플랫폼이다. 2020년, Pickle Finance는 악의적인 컨트랙트 호출로 인해 자금을 탈취당하는 사건이 발생하였다.

공격자는 Pickle Finance의 스마트 컨트랙트에서 발생하는 복잡한 상태 전환을 악용하여 공격하였다. 이 과정에서 컨트랙트의 상태를 조작하여, 자산을 안전하게 보호할 수 있는 메커니즘을 무력화시켰다. 이로 인해 약 2,000만 달러 이상의 자산이 탈취되었다.

10. Uranium Finance 해킹

Uranium Finance는 탈중앙화된 거래소(DEX)로, 사용자가 다양한 암호화폐를 거래할 수 있는 플랫폼이다. 2021년, Uranium Finance는 스마트 컨트랙트를 업데이트하는 과정에서 발생한 오류로 인해 큰 손실과 피해를 입게 되었다.

Uranium Finance 공격은 스마트 컨트랙트 업데이트 중 발생한 코드 오류가 공격의 원인이 되었다. 이 오류를 악용한 공격자는 약 5,000만 달러 이상의 자산을 탈취하였으며. 업데이트 과정에서 발견된 취약점은 새로운 기능 추가 또는 기존 기능 수정 시 발생할 수 있는 잠재적 위험성을 보여주었다.

11. Poloniex 해킹 (2024)

Poloniex는 암호화폐 거래소로, 2024년 초 스마트 컨트랙트의 취약점을 악용한 공격으로 약 1억 2천 6백만 달러 이상의 자산이 탈취되는 사건이 발생했다. 이 해킹은 블록체인 기술의 기술적 결함이 얼마나 심각한 결과를 초래할 수 있는지를 잘 보여주었다. 해커는 Poloniex의 스마트 컨트랙트 코드에서 발견된 취약점을 이용해 대규모 자산을 불법으로 이체하는 데 성공했다. 이 사건은 특히 스마트 컨트랙트의 설계 및 구현 단계에서 발생할 수 있는 보안 취약점이 얼마나 큰 자산 피해로 이어질 수 있는지에 대한 사례로 남았다.

Poloniex 해킹은 블록체인 생태계 내에서 보안 강화의 필요성을 부각시키는 계기가 되었으며, 이와 같은 사건들이 더 이상 발생하지 않도록 하기 위해서는 지속적인 보안 점검과 코드 리뷰를 필수적으로 검토해야 한다는 중요한 사례가 되었다.

12. HTX Exchange 해킹 (2024)

2024년 3월, HTX Exchange는 약 8천 5백만 달러 상당의 암호화폐 자산이 탈취되는 심각한 해킹 사건이 발생하였다. 이번 사건은 중앙화된 거래소도 해커들의 주요 표적이 될 수 있음을 다시 한 번 보여주는 사례로, 보안 강화의 필요성을 절실히 대두되었다. 해커들은 HTX Exchange의 보안 시스템에서 발견된 취약점을 악용하여, 플랫폼에 보관 중이던 대규모 자산을 탈취하는 데 성공하였다.

이 사건은 중앙화된 거래소가 보안 위협에 얼마나 취약할 수 있는지를 보여줬으며, 특히 사용자의 자산이 중앙화된 시스템에 집중되어 있는 경우 이러한 위험이 더욱 크다는 것을 부각시켰다. HTX Exchange 해킹은 중앙화된 플랫폼이 보안 시스템을 지속적으로 개선하고, 해커들의 공격에 대비하기 위해 보다 강력한 보안 조치를 도입해야 함을 시사 하였다.

이와 같은 해킹 사건은 중앙화된 거래소뿐만 아니라 전체 암호화폐 생태계에 심각한 영향을 미칠 수 있으며, 사용자들이 자신의 자산을 안전하게 보호하기 위해 중요한 검토가 필요하다는 것을 부각시키는 사건이 되었다.

13 Kronos Research 해킹 (2024)

2024년, Kronos Research는 약 4천 8백만 달러 상당의 자산이 탈취된 해킹 사건의 피해를 입었다. 이 사건은 스마트 컨트랙트와 블록체인 기술의 보안 취약점이 어떻게 악용될 수 있는지를 잘 보여주는 사례로, 해커들은 Kronos Research의 시스템에서 발견된 보안 허점을 이용해 자산을 탈취하였다.

Kronos Research는 알고리즘 트레이딩과 연구를 전문으로 하는 블록체인 기반 플랫폼으로, 기술적으로 고도화된 블록체인 환경에서도 보안 취약점이 존재할 수 있다는 것을 부각 시키는 사건이 되었다. 이번 사건은 특히 DeFi(탈중앙화 금융)와 같은 고도화된 금융 서비스에서도 보안 강화를 위한 지속적인 노력이 필요함을 시사하였다.

블록체인 기술이 발전하고 있는 현재의 환경에서도 해커들이 시스템의 취약점을 노려 대규모 자산을 탈취할 수 있음을 확인시켜주며, 더욱 강화된 보안 조치와 지속적인 보안 점검이 필요하다는 점을 일깨워 주는 사건이다.

14 Qbit Finace 해킹 사건 (2023)

Qubit Finance 해킹 사건은 2022년 1월 말에 발생한 주요 디파이(DeFi) 해킹 사고 중 하나로, 약 8천만 달러 상당의 암호화폐가 도난당한 사건이다. 이 사건은 Qubit Finance의 QBridge 프로토콜을 타겟으로 발생하였다. QBridge는 이더리움 블록체인과 바이낸스 스마트 체인(BSC) 간의 자산 교환을 가능하게 하는 크로스체인 브릿지이다.

해커는 Qubit의 스마트 계약 코드에 존재하는 논리적 취약점을 악용하였다. 구체적으로, 해커는 이더리움을 실제로 입금하지 않고도 입금된 것처럼 시스템을 속여, 77,162개의 가짜 qXETH 토큰을 생성하였다. 이 가짜 토큰을 담보로 206,809개의 바이낸스 코인(BNB)을 QBridge에서 인출할 수 있었다. 결과적으로, 해커는 약 8천만 달러 상당의 자산을 탈취했다.

QBrdige는 이더리움과 바이낸스 스마트 체인(BSC)간의 자산 교환을 지원하는 크로스체인 브릿지로 설계되었다. 이 시스템을 통해 사용자는 ERC-20과 BEP-20 토큰을 손쉽게 교환할 수 있었다. 이것을 통해 Qbit Finance의 스마트 계약에는 치명적인 논리 결함이 있다는 것을 확인 할 수 있었다.

공격자는 이 스마트 계약 코드에 있는 오류를 악용했다. 이 오류로 인해, 해커는 실제로 이더리움을 입금하지 않았음에도 불구하고 시스템이 입금이 완료된 것으로 잘못 인식하도록 만들었다. 이러한 허점을 이용해 해커는 가짜 자산을 생성할 수 있었고, 이를 기반으로 실제 자산을 인출할 수 있었다.

사건 발생 후, Qubit Finance 팀은 해커에게 버그 바운티 프로그램을 제안하며 자산 반환을 요청했다. 그러나 이 요청은 해커 측에서 승낙 하지 않았다. 이 사건으로 인해 Qubit의 토큰 가치가 급락했고, 디파이 생태계에서 크로스체인 브릿지의 보안 문제에 대한 경각심이 높아졌다.

4. 위협완화 방법론

1. 소프트웨어 관점 에서의 위협 완화 방법론

가) CWE (Common Weakness Enumeration)

CWE는 소프트웨어 보안 약점을 체계적으로 분류하고 관리하기 위해 만들어진 표준 목록이다. 이 목록은 소프트웨어 개발 과정에서 발생할 수 있는 다양한 보안 약점을 식별하고 분류하는데 도움 주기 위해 고안되었다. CWE는 소프트웨어 보안을 책임지는 개발자와 보안 전문가들이 공통적으로 활용할 수 있는 중요한 도구로, 소프트웨어의 안전성을 높이기 위해 필수적인 역할을 한다.

CWE의 핵심 목적은 소프트웨어에서 발생할 수 있는 보안 약점을 명확히 정의하고 이러한 약점이 시스템에 미치는 영향을 이해하며 이를 예방하거나 완화하는 방법을 제시하는 것이다. CWE는 소프트웨어 구조의 결함이나 설계상의 문제로 인해 발생할 수 있는 다양한 약점을 포괄적으로 다루고 있다. 예를 들어 버퍼 오버플로우, SQL 인잭션, 크로스 사이트 스크립트증과 같은 취약점 들이 포함 되며, 이러한 취약점 들이 시스템 보안에 어떤 영향을 미칠 수 있는지 상세히 설명한다.

CWE는 각 취약점에 고유한 식별자를 부여하고, 해당 취약점이 발생하는 원인과 공격 방법, 이를 방지하기 위한 조치를 포함한 상세한 정보를 제공하는데, 이를 통해 개발자들은 특정 취약점을 이해하고, 이를 방지하거나 해결하기 위한 적절한 대응책을 마련할 수 있는 기회를 제공한다. CWE 목록은 정기적으로 계속 업데이트되며, 새로운 보안 약점이 등장할 때 이를 반영하여 약점에 대한 정보를 꾸준히 공유하고 있다.

CWE는 다양한 방식으로 활용될 수 있는 큰 장점이 있다. 예를 들어, 개발자들은 CWE를 통해 보안 약점을 예방하는 코딩 방식을 할수 있으며, 보안 스캐너와 같은 진단 도구는 CWE를 참고하여 소프트웨어의 취약점을 식별하고 이를 개선할 수 있다. 보안 감사 과정에서도 CWE는 시스템의 보안 상태를 평가하는 중요한 기준으로 사용되고 있다. 또한, CWE는 CVE(Common Vulnerabilities and Exposures) 같은 다른 보안 표준과 연계되어, 시스템에서 발생하는 취약점의 구체적인 발생 원인과 그에 대한 대응 방안을 설명하는 데 도움을 주고 있

다. CWE는 정량적으로 CWSS(Common Weakenss Scoring System) 으로 점수를 산출 할 수 있다.

CWE의 단점은. 모든 시스템이 동일하게 CWE를 적용할 수 있는 것은 아니며, 시스템의 특성과 상황에 따라 약점이 다르게 나타날 수 있다. 또한, CWE 목록은 계속해서 업데이트되지만, 모든 새로운 약점을 즉시 반영하는 것은 쉽지 않다. 그럼에도 CWE는 소프트웨어 보안의 중요한 요소로 자리 잡고 있으며, 이를 통해 개발자와 보안 전문가들은 보다 안전한 시스템을 구축하고 유지하는 데 큰 도움을 받을 수 있다.

CWE는 다음과 같은 구성으로 되어있다:

1. **CWE ID:** 각 CWE 항목은 고유한 식별자 ID를 가지고 있다. 이 ID는 CWE 목록 내에서 보안약점을 식별하는 데 사용되며 보안 보고서 진단도구, 교육 자료 등에서 일괄되게 사용하고 있다.

2. **약점이름 (Weakness Name)**: CWE 항목은 그 약점을 명확히 나타내는 이름을 가지고 있다. 이 이름은 해당 약점의 핵심 문제를 간결하게 설명하며, 약점의 본질을 쉽게 이해할 수 있도록 한다

3. **약점 설명(Description)**: CWE 항목에는 해당 약점이 무엇인지에 대한 상세 설명이 포함되고 있다. 이 설명은 약점이 발생한 이유, 시스템에 미치는 영향, 일반적인 발생 시나리오 등을 다루고 있다.

4. **약점의 발생 원인 (Cause)**:약점이 발생하는 원인에 대한 정보가 제공되고 있다. 이는 소프트웨어 설계나 구현 단계에서의 실수, 잘못된 설정, 불충분한 입력 검증 등과 같은 원인이 포함되어 있다.

5. **공격 방법(Attacker's Technique)**: 해당 약점을 공격자가 어떻게 악용할 수 있는지에 대한 정보가 포함되어있다. 이 섹션은 공격자가 약점을 이용해 시스템에 침입하거나 데이터를 탈취하는 방법을 설명하고 있다.

6. **예제(Examples)**: 실제 코드 예제나 시나리오가 제공되어, 해당 약점이 어떻게 나타날 수 있는지를 구체적으로 설명함. 이를 통해 개발자는 자신의 코드에서 유사한 문제가 발생할 수 있는지를 확인할 수 있다.

7. **완화 방안(Mitigation)**: 약점을 방지하거나 완화하는 방법에 대한 권장 사항이 포함되어있다. 이 섹션에서는 개발자들이 코드 작성, 시스템 설계, 또는 배포 단계에서 약점을 피하거나 그 영향을 줄일 수 있는 방법을 설명하고 있다.

8. **관련 CWE(References/Relationships)**: 해당 약점과 관련된 다른 CWE 항목들에 대한 링크나 참조가 제공되고 있다. 이는 관련 약점들이 어떻게 연결되는지를 이해하고, 종합적인 보안 전략을 수립하는 데 도움이 된다.

9.**부가 정보(Additional Information)**: 해당 약점과 관련된 추가 정보나 외부 링크가 포함될 수 있다. 예를 들어, 심층적인 연구 논문, 도구, 또는 보안 모범 사례에 대한 참조가 제공될 수 있다.

나) CVE (Common Vulnerability Enumeration)

CVE(Common Vulnerabilities and Exposures)는 전 세계적으로 공개된 소프트웨어 보안 취약점을 표준화된 방식으로 식별하고 관리하기 위해 만들어진 목록이다. 이 시스템은 특정 보안 취약점을 명확히 정의하고, 이를 관리하는 데 도움을 주기 위해 고안되었다. CVE는 보안을 주로 일하는 보안팀, 소프트웨어 개발팀, 기업, 그리고 보안 도구 개발자들이 공통적으로 활용할 수 있는 도구로, 보안 문제를 신속하게 파악하고 대응하는 데 필수적인 역할을 한다.

CVE의 핵심 목적은 소프트웨어나 하드웨어에서 발생할 수 있는 보안 취약점을 명확히 식별하고, 이러한 취약점이 시스템에 미치는 영향을 이해하며, 이를 해결하기 위한 조치를 제공하는 것이다. CVE는 특정 소프트웨어 버전이나 시스템 구성에서 발생할 수 있는 다양한 보안 문제를 포괄적으로 다룬다. 예를 들어, 취약한 인증 메커니즘, 버퍼 오버플로우, SQL 인젝션과 같은 취약점들이 포함되며, 이러한 문제들이 시스템 보안에 미칠 수 있는 영향을 상세히 설명한다.

CVE 목록에서 각 취약점은 고유한 식별자인 CVE ID를 부여받는다. 이 식별자는 "CVE-연도-번호" 형식으로 표시되며, 예를 들어 "CVE-2023-12345"와 같이 특정 취약점을 명확히 지칭한다. 각 CVE 항목에는 해당 취약점이 발생하는 원인과 공격자가 이를 악용할 수 있는 방법, 그리고 이를 방지하기 위한 권장 조치가 포함된 상세한 정보가 제공된. CVE 목록은 정기적으

로 업데이트되며, 새로운 취약점이 발견될 때 이를 빠르게 반영한다.

CWE는 다음과 같은 구성으로 되어있다:

1. **CVE ID**: CVE ID는 각 보안 취약점에 고유하게 부여되는 식별 아이디 이다. 이 식별 아이디는 "CVE-연도-번호" 형식으로 표시되며, 예를 들어 "CVE-2023-12345"와 같은 형태로 나타낼수 있다. CVE ID는 특정 취약점을 명확히 식별하고, 다양한 보안 보고서, 연구 문서, 보안 도구에서 동일한 취약점을 일관되게 참조할 수 있다.

3. **약점 설명(Description)**: CVE 항목에는 해당 CVE 항목이 다루는 보안 취약점에 대한 간결한 요약을 제공한다. 이 설명은 취약점이 발생하는 상황, 시스템에 미치는 영향, 그리고 공격자가 이를 어떻게 악용할 수 있는지에 대해 간략히 설명한다. 이 정보는 취약점의 본질을 이해하는데 중요한 역할을 한다.

4. **상태 (Statue)**: 상태는 해당 CVE 항목의 검토 및 승인 과정에서의 현재 상태를 나타낸다. CVE 항목은 "Candidate(후보)" 상태로 시작하며, 이후 전문가 검토를 거쳐 "Entry(정식 등록)" 상태로 변경된다. 이 상태 정보는 CVE 항목이 현재 어느 단계에 있는지를 이해하는 데 도움을 준다.

5. **심각도 평가(Severity Rating)**: CVE 항목 자체에는 심각도 평가 점수가 포함되지는 않지만, CVE 항목은 종종 CVSS(Common Vulnerability Scoring System) 점수와 함께 사용된다. CVSS는 해당 취약점이 얼마나 심각한지를 평가하여 점수로 표현하며, 이 점수는 보안 대응의 우선순위를 결정하는 데 활용될수 있다.

6. **공개 날짜(Publication Date)**: 공개 날짜는 해당 CVE 항목이 처음 공개된 날짜를 나타낸다. 이 정보는 취약점이 언제 발견되었고, 얼마나 오랫동안 노출되었는지를 이해하는 데 도움을 준다.

다) SWC Registry (Smart Contract Weakness Classification and Test Cases)

SWC Registry(Smart Contract Weakness Classification and Test Cases)는 스마트 컨트랙트의 보안 취약점을 체계적으로 분류하고 관리하기 위해 개발된 표준화된 목록이다. 이 레지스트리는 스마트 컨트랙트에서 발생할 수 있는 취약점을 식별하고, 이러한 취약점이 시스템에 미치는 영향을 분석하며, 이를 해결하기 위한 지침을 제공하는 것을 목표로 하고 있다. 블록체

인 기술, 특히 이더리움과 같은 스마트 컨트랙트 플랫폼이 널리 사용됨에 따라, 스마트 컨트랙트의 안전성을 보장하는 것이 매우 중요해졌다. SWC Registry는 이러한 문제를 해결하기 위해 고안되었으며, 개발자와 보안 전문가들이 스마트 컨트랙트의 보안을 강화하는 데 중요한 도구로 활용되고 있다.

스마트 컨트랙트는 블록체인 상에서 자동으로 실행되는 계약으로, 금융 거래, 자산 관리, 탈중앙화 애플리케이션(DApps) 등 다양한 분야에서 현저히 사용되고 있다. 하지만 스마트 컨트랙트는 코드에 내재된 오류나 취약점으로 인해 심각한 문제를 초래하고 있다. 이러한 이유로 스마트 컨트랙트의 보안은 블록체인 기술의 성공적인 도입과 운영을 위해 매우 중요하다. 기존의 소프트웨어 보안 관점에서 CWE(Common Weakness Enumeration)와 같은 표준화된 목록이 사용되어 왔지만, 스마트 컨트랙트의 특수성을 반영한 보안 프레임워크가 필요하다는 인식이 생겼고. 그 인식으로 인해 SWC Registry는 스마트 컨트랙트의 특화된 보안 약점을 체계적으로 분류하고 관리하기 위해 설계되었다. SWC Registry는 스마트 컨트랙트에 특화된 보안 위협을 다루며, 이러한 취약점을 이해하고 해결하는 데 필요한 구체적인 정보를 제공하고 있다.

SWC Registry의 주요 목적은 스마트 컨트랙트에서 발생할 수 있는 보안 취약점을 명확히 정의하고, 이를 체계적으로 분류하며, 이러한 취약점이 시스템에 미치는 영향을 이해하고 대응 방안을 제시하는 것이다. 이를 통해 개발자는 코드 작성 시 잠재적인 위험 요소를 사전에 인지하고, 이를 예방할 수 있다. 또한, SWC Registry는 보안 취약점에 대한 표준화된 용어와 분류 체계를 제공하여, 개발자와 보안 전문가들이 공통된 언어를 사용하여 취약점을 논의하고 해결할 수 있도록 돕는다. 이 목록은 또한 교육 자료로 활용될 수 있으며, 개발자들이 스마트 컨트랙트의 보안성을 높이기 위한 코딩 방법을 학습할 수 있는 도구로 사용될 수 있다. SWC Registry는 각 취약점에 대해 구체적인 테스트 케이스를 제공하여, 개발자가 자신의 스마트 컨트랙트 코드에서 이러한 취약점이 발생하지 않도록 검증하는 데 도움을 준다.

SWC Registry는 각 보안 취약점에 대해 고유한 식별자(SWC ID)를 부여하고, 이를 상세히 설명하는 구조로 이루어져 있다. 각 SWC ID는 특정 취약점을 명확히 식별하고, 이를 참조할 수 있도록 도와준다. 각 항목에는 해당 취약점을 설명하는 이름, 취약점의 발생원인, 이를 방지하거나 완화할 수 있는 방법, 그리고 관련된 테스트 케이스와 추가 정보에 대한 링크가 포

함되어 있다. 이러한 구성 요소들을 통해 SWC Registry는 스마트 컨트랙트 보안을 체계적으로 관리하고, 스마트 컨트랙트의 안전성을 높이는 데 중요한 역할을 한다.

SWC Registry는 스마트 컨트랙트 개발 과정에서 다양한 방식으로 활용될 수 있습니다. 개발자는 이를 참고하여 코드 작성 시 보안 취약점을 사전에 인지하고, 이를 예방하는 코딩 방법을 채택할 수 있다. 보안 전문가들은 SWC Registry를 사용하여 스마트 컨트랙트의 보안성을 평가하고, 발견된 취약점에 대한 적절한 대응 방안을 제시할 수 있다. 이 레지스트리는 보안 감사 과정에서도 중요한 역할을 하며, 스마트 컨트랙트의 보안 상태를 평가하고, 잠재적인 약점이 얼마나 잘 관리되고 있는지를 분석하는 데 사용된다.

SWC Registrary는 다음과 같은 구성으로 되어있다:

1. **약점 이름(Weakness Name)**: SWC 항목은 해당 취약점을 명확하게 설명하는 이름을 가지고 있다. 이 이름은 취약점의 핵심 문제를 간결하게 전달하며, 취약점의 본질을 쉽게 이해할 수 있도록 돕는다.
2. **약점 설명(Description)**: SWC 항목에는 해당 취약점이 무엇인지에 대한 상세한 설명이 포함된다. 이 설명은 취약점이 발생하는 이유, 시스템에 미치는 영향, 그리고 발생할 수 있는 일반적인 상황을 설명한다.
3. **약점의 발생 원인(Cause)**: 취약점이 발생하는 원인에 대한 정보가 제공되며, 이 섹션에서는 소프트웨어 설계 과정이나 구현 단계에서의 오류, 잘못된 설정, 불충분한 입력 검증 등과 같은 원인을 설명한다.
4. **공격 방법(Attacker's Technique)**: 해당 취약점을 공격자가 어떻게 악용할 수 있는지에 대한 정보가 제공된다. 이 섹션에서는 공격자가 취약점을 이용해 시스템에 침투하거나 데이터를 탈취하는 방법을 설명한다.
5. **예제(Examples)**: 실제 코드 예제나 시나리오가 제공되어, 해당 취약점이 어떻게 나타날 수 있는지를 구체적으로 설명된다. 이를 통해 개발자는 자신의 코드에서 유사한 문제가 발생할 가능성을 확인할수 있다.

라) EEA Security Levels (Ethereum Enterprise Alliance Security Levels)

EEA Security Levels(Ethereum Enterprise Alliance Security Levels)는 이더리움 기반 블록체

인 솔루션의 보안을 평가하고 분류하기 위한 표준화된 프레임워크다. 이 프레임워크는 스마트 컨트랙트와 탈중앙화 애플리케이션(DApps)의 보안성을 체계적으로 분석하고, 기업들이 자신들의 블록체인 솔루션이 얼마나 강력한 보안 체계를 갖추고 있는지를 평가할 수 있도록 정보를 제공해주는 역할을 하고 있다. 블록체인 기술의 활용이 점점 확대됨에 따라, 이러한 시스템의 보안을 보장하는 것이 필수적이 되었으며, EEA Security Levels는 이 과제를 해결하기 위한 중요한 도구로 자리 잡고 있다.

이더리움과 같은 블록체인 플랫폼이 널리 사용됨에 따라, 스마트 컨트랙트가 수행하는 자동화된 금융 거래와 같은 중요한 기능에서의 보안성은 매우 중요한 문제로 대두되었는데. 스마트 컨트랙트의 보안 취약점은 치명적인 재정적 손실이나 시스템 장애를 초래할 수 있기 때문에, 이러한 시스템의 보안을 평가하고 강화할 필요가 커졌다. EEA Security Levels는 이러한 필요성에 따라 개발되었으며, 기업들이 블록체인 솔루션의 보안 강도를 명확히 평가하고, 필요한 개선 조치를 취할 수 있는 체계적인 방법을 제공한다.

EEA Security Levels의 주요 목적은 블록체인 기반 솔루션에서 발생할 수 있는 보안 위협을 체계적으로 평가하고, 이러한 위협에 대비할 수 있는 표준화된 방법을 제공하는 것이다. 이 표준은 시스템의 보안 수준을 평가하여 기업들이 자신들의 블록체인 솔루션이 어떤 보안 수준에 해당하는지를 명확히 파악할 수 있도록 돕는다. 또한, EEA Security Levels는 보안 평가 결과를 기반으로 기업들이 보안을 강화하기 위한 구체적인 지침을 제공한다. 이를 통해 다양한 블록체인 솔루션 간의 보안 수준을 비교할 수 있으며, 일관된 방법으로 보안을 관리할 수 있는 표준화된 기준을 제공한다. 이러한 표준화된 보안 수준 평가와 지침을 통해 기업 간의 신뢰성이 높아지며, 블록체인 솔루션의 채택이 촉진될 수 있다.

EEA Security Levels는 다양한 보안 요구사항을 반영하여 여러 단계의 보안 레벨로 구성된다. 각 보안 레벨은 시스템이 충족해야 하는 특정 보안 요구사항을 정의하며, 레벨이 높을수록 더 강력한 보안 체계를 의미한다. 이 프레임워크는 스마트 컨트랙트 코드의 검토, 보안 감사, 접근 제어 메커니즘, 데이터 암호화와 같은 구체적인 보안 요구사항을 포함하며, 이러한 요구사항을 충족하는지 여부에 따라 보안 수준이 평가된다. 또한, EEA Security Levels는 표준화된 평가 프로세스를 통해 시스템의 보안 상태를 종합적으로 평가하며, 이를 문서화하여 필요에 따라 인증을 받을 수 있도록 지원한다. 이러한 보안 평가 결과는 시스템의 보안 상태를 명확

히 보고하고, 기업 간의 신뢰를 구축하는 데 중요한 역할을 한다.

EEA Security Levels는 기업들이 블록체인 솔루션의 보안성을 평가하고 강화하는 데 광범위하게 활용될 수 있다. 이 프레임워크를 통해 기업들은 자신들의 솔루션이 어떤 보안 수준에 해당하는지를 명확히 이해할 수 있으며, 이를 기반으로 보안 전략을 수립할 수 있다. 또한, EEA Security Levels는 기업 간의 신뢰성을 높이는 데 중요한 역할을 한다. 보안 수준이 명확히 정의되고, 이를 기반으로 한 인증을 통해 기업 간의 협력과 거래가 더욱 안전하게 이루어질 수 있다. 따라서 EEA Security Levels는 블록체인 기술의 도입과 운영에서 중요한 역할을 하고 있다.

EEA Security Levels는 다음과 같은 구성으로 되어있다:

1.보안 레벨 정의(Security Level Definitions): 보안 레벨 정의는 시스템의 보안 수준을 평가하기 위한 기본적인 기준을 제공한다. 각 보안 레벨은 시스템이 충족해야 하는 특정 보안 요구사항을 나타내며, 레벨이 높아질수록 더 엄격한 보안 기준을 적용받는다. 예를 들어, 낮은 레벨에서는 기본적인 보안 검토가 요구될 수 있지만, 높은 레벨에서는 더욱 철저한 보안 검토와 다중 레이어의 보안 통제가 필요하다.

2. 보안 요구사항(Security Requirements): 보안 요구사항은 각 보안 레벨에 해당하는 구체적인 조건을 정의한다. 이 요구사항에는 스마트 컨트랙트의 코드 검토, 보안 감사, 데이터 암호화, 접근 제어 등의 보안 요소가 포함된다. 이 요구사항을 충족함으로써 시스템은 해당 보안 레벨에 부합하는 보호를 제공할 수 있으며, 이는 시스템의 전체적인 보안성을 높이는 데 기여한다.

3. 평가 프로세스(Security Assessment Process): 평가 프로세스는 시스템의 보안 수준을 측정하고 검증하기 위한 표준화된 절차를 제공한다. 이 과정은 보안 감사, 코드 리뷰, 침투 테스트, 취약점 분석 등을 포함하며, 이를 통해 시스템이 설정된 보안 요구사항을 충족하는지 여부를 검증한다. 이 프로세스는 보안 평가의 일관성을 유지하고, 다양한 시스템 간의 보안 수준을

비교할 수 있도록 한다.

4. 인증 및 보고(Certification and Reporting): 인증 및 보고는 보안 평가 결과를 문서화하고, 필요한 경우 이를 기반으로 인증을 제공하는 단계이다. 이 단계에서는 시스템이 특정 보안 레벨을 충족했음을 공식적으로 증명할 수 있으며, 이를 통해 외부 이해관계자에게 시스템의 보안 상태를 명확히 전달할 수 있다. 인증된 보안 상태는 기업 간의 신뢰를 높이고, 블록체인 솔루션의 채택을 촉진할 수 있는 중요한 요소가 된다.

5. 지속적인 모니터링과 업데이트(Continuous Monitoring and Updates): 지속적인 모니터링과 업데이트는 시스템의 보안 상태를 유지하고 개선하기 위한 중요한 과정이다. 블록체인 기술과 보안 위협이 계속해서 진화함에 따라, 시스템의 보안 상태를 지속적으로 모니터링하고 새로운 위협에 대응하기 위해 보안 정책과 절차를 정기적으로 업데이트해야 한다. 이를 통해 시스템은 항상 최신의 보안 표준을 유지하며, 새로운 보안 요구사항에 적절히 대응할 수 있다.

5. SSDLC(Secure Software Devlopment Lifecycle)

1.SSDLC(Secure Software Devlopment Lifecycle)

SSDLC (Secure Software Development Life Cycle)은 소프트웨어 개발 과정에서 보안 요소를 체계적으로 통합하여, 소프트웨어의 전반적인 개발 단계에서 발생할 수 있는 보안 문제를 예방하고 지속적으로 관리할 수 있도록 하는 방법론이다. 이 방법론은 기존의 소프트웨어 개발 라이프사이클(SDLC)에 보안 내재화를 추가하여, 개발 초기부터 보안 요구사항을 명확히 식별하고 내재화 하는 것을 목표로 한다.

이 과정은 소프트웨어 개발의 각 단계에서 보안을 내재화하는 것을 강조하며, 특히 스마트 컨트랙트와 같은 고신뢰성 시스템의 개발에 필수적인 접근 방식을 제시한다. SSDLC는 개발 초기 단계에서 보안 요구사항을 식별하고 이를 설계, 구현, 테스트, 배포 및 운영 단계에서 지속적으로 반영함으로써, 소프트웨어의 전 주기에서 보안이 강화되도록 하는 것이 목표이다. 이를 통해, 개발된 소프트웨어가 보안 문제에 위협이 되지 않고 안정적으로 운영될 수 있도록 보장한다. 또한 보안을 내재화를 요구사항 분석 단계부터 적용함으로써 소프트웨어를 완성하고 보안을 내재화 하는 것보다 비용적 측면에서 효율적이다.

이 방법론은 DevOps와 DevSecOps의 원칙을 기반으로 하며, 개발 주기의 모든 단계에서 보안 활동이 통합되도록 설계되어 있다. SSDLC는 특히 스마트 컨트랙트와 같은 시스템에서 높은 수준의 보안과 신뢰성을 달성하기 위한 최적의 조건을 가지고 있는 방법론이다.

2.DevSecOps

DevOps는 소프트웨어 개발과 운영을 통합하여, 더 빠르고 안정적으로 소프트웨어를 배포할 수 있도록 설계된 방법론이다. 이 접근법은 개발팀과 운영팀 간의 협업을 강화하여, 소프트웨어 개발 과정에서 발생할 수 있는 문제를 미리 해결하는 데 중점을 둔다. 이를 통해 소프트웨어의 개발 속도와 품질을 동시에 높일 수 있다. DevOps의 주요 원칙 중 하나는

자동화이다. 이는 빌드, 테스트, 배포, 모니터링 등 소프트웨어 개발 주기의 모든 단계를 자동화함으로써, 효율성을 극대화하고 인적 오류를 줄이는 것을 목표로 한다. 또한 지속적 통합(CI)과 지속적 배포(CD)라는 원칙을 통해, 소스 코드의 변경 사항을 자주 통합하고, 자동화된 테스트를 통해 안정성을 확보한 후, 즉시 배포하는 프로세스를 구현한다. 이렇게 함으로써 새로운 기능이나 수정 사항이 빠르게 사용자에게 전달될 수 있다.

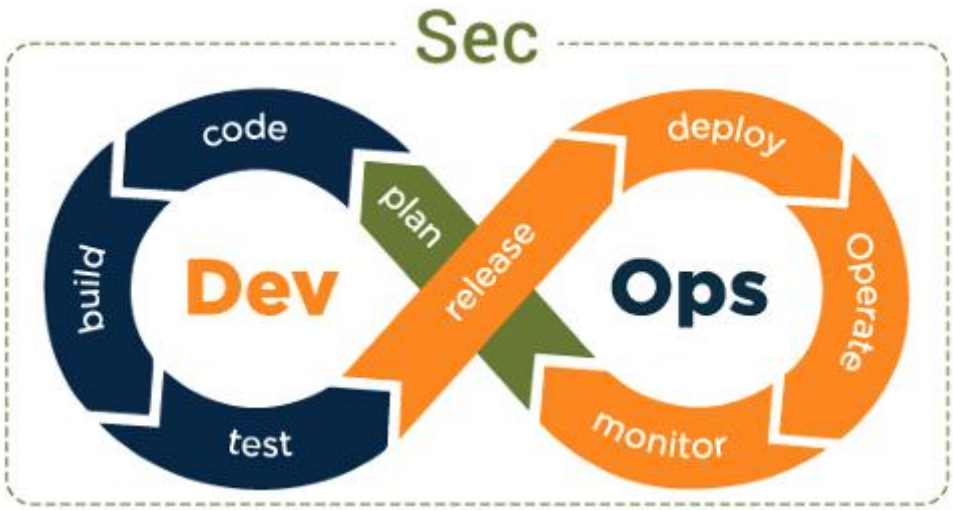

그림 1. DevOps 와 DevSecOps의 구상도

한편, DevSecOps는 DevOps의 원칙에 보안을 추가한 개념으로, 소프트웨어 개발 초기 단계부터 보안을 통합하는 것을 목표로 한다. 이를 통해 소프트웨어 개발의 모든 단계에서 보안 활동이 이루어지도록 한다. DevSecOps의 중요한 원칙 중 하나는 'Shift Left'로, 보안 활동을 개발 주기의 초기 단계로 이동시켜, 코드 작성 단계에서부터 보안 문제를 식별하고 해결하는 것이다. 이를 통해 배포 후에 발견될 수 있는 보안 결함을 방지하고, 수정 비용을 최소화할 수 있다. 또 다른 중요한 원칙은 자동화된 보안 테스트입니다. DevOps의 자동화 원칙을 따르며, 코드의 보안성을 지속적 통합(CI) 과정에서 자동으로 검증하고, 발견된 취약점에 대해 즉각적인 피드백을 제공한다. 마지막으로, DevSecOps는 보안에 대한 팀의 인식과 교육을 중시하여, 개발팀과 운영 팀이 보안 원칙과 최신 위협에 대해 지속적으로 학습할 수 있는 교육 프로그램을 마련한다. 이를 통해 팀 전체가 보안을 고려한 코딩과 운영을 수행할 수 있도록 한다.

3.SSDLC에서 정의하는 각 개발 단계

SSDLC는 다음과 같은 소프트웨어 개발단계에 대해 명확히 정의하고 있다.

A. 계획(Planning) 단계: 이 단계에서는 소프트웨어 개발 프로젝트의 범위와 목적을 정의할 뿐만 아니라, 보안 요구사항도 함께 설정한다. 이때 GDPR과 같은 법적 요구사항이나 산업 표준을 체크리스트로 관리하여, 개발 과정에서 준수해야 할 보안 규정을 명확히 한다.

B. 요구사항 정의(Requirements) 단계: 소프트웨어의 기능적 요구사항뿐만 아니라, 보안 요구사항도 명확히 정의한다. 이 과정에서는 과거의 보안 사고를 분석하여, 유사한 보안 리스크를 사전에 차단하기 위한 예방 조치를 마련한다. 스마트 컨트랙트의 경우, 사고 패턴을 기반으로 잠재적 위험을 미리 식별하는 것이 중요하다.

D. 설계(Design) 단계: 보안 요구사항을 반영한 소프트웨어 아키텍처를 설계합니다. 보안 아키텍처 설계에서는 권한 관리, 데이터 암호화, 인증 및 접근 제어 등의 요소를 고려해야 한다. 이 단계에서는 또한 보안 결함이 최소화될 수 있도록 소프트웨어 구성 요소를 설계한다.

E. 구현(Implementation) 단계: 시큐어 코딩 가이드를 적용하여 보안이 내재된 코드를 작성한다. 스마트 컨트랙트의 경우, 트랜잭션 처리의 안전성과 보안을 확보하기 위해, 코드 작성 시 시큐어 코딩 가이드라인을 철저히 준수하는 것이 중요하다. 이러한 가이드는 잠재적인 보안 취약점을 최소화하는 데 도움을 준다.

F. 테스트 및 검증(Testing and Validation) 단계: 정적 분석 및 동적 분석 도구를 사용하여 코드의 보안 취약점을 검사한다. 정적 분석은 코드를 실행하지 않고 구조적 결함을 찾아내는 데 중점을 두며, 동적 분석은 소프트웨어를 실제로 실행하면서 발생할 수 있는 잠재적인 보안 문제를 탐지한다. 스마트 컨트랙트의 경우, 이러한 검증 단계에서 제로데이 취약점을 예방하는 것이 매우 중요하다.

G. 배포(Deployment) 단계: 소프트웨어를 실제 운영 환경에 배포할 때, 보안 설정이 제대로 적용되었는지 확인한다. 특히, 스마트 컨트랙트는 한번 배포되면 수정이 어려우므로, 이 단계에서의 검증이 더욱 중요하다.

H. 운영 및 모니터링(Operation and Monitoring) 단계: 소프트웨어가 운영되는 동안, 지속적인 모니터링을 통해 보안 상태를 유지한다. DApp 모니터링 도구를 통해 스마트 컨트랙트의 실시간 상태를 감시하고, 비정상적인 활동이 감지되면 즉시 대응한다.

4.SSDLC에서 정의하는 보안내재화

SSDLC에서 위협모델링은 소프트웨어 개발 과정에서 잠재적인 보안 위협을 사전에 식별하고 이를 반영하여, 개발 초기 단계부터 보안을 강화하는 절차이다. 이 모델링 작업은 요구사항 정의(Requirements)와 설계(Design) 단계에서 작업이 이루어져야 한다. 이 단계에서 위협 을 수행하면, 소프트웨어는 기능적 요구사항뿐만 아니라 보안 요구사항을 구체화할 때 발생할 수 있는 보안 위험 요소를 미리 파악하고 대응할 수 있다. 이 프로세스를 통해 결과적으로 보안 리스크를 최소화할수 있고, 또한 안전한 소프트웨어 아키텍처를 설계하는 데 기여한다. 또한, 보안내재화는 소프트웨어 개발의 초기 단계에서 수행될 때 가장 효과적이다. 초기 단계에서 잠재적 보안 문제를 식별하고, 이에 대한 해결책을 마련하면, 이후 개발 과정에서 발생할 수 있는 보안 문제를 미리 방지할 수 있다. 특히, 스마트 컨트랙트와 같은 고신뢰성 시스템의 경우, 한 번 배포되면 수정이 어려운 특성을 가지고 있기 때문에 초기 단계에서 철저한 보안 검토가 필수적이다. 따라서 개발 초기부터 체계적으로 체크리스트를 수행하는 것은 이러한 시스템의 보안성과 신뢰성을 달성하는데 중요한 프로세스 이다.

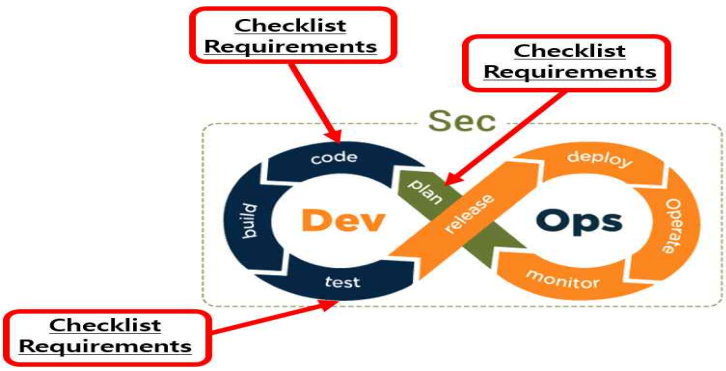

그림 2. Check List Requirement in DevSecOps

각 단계에서의 고려해야할 구체적인 보안 점검 활동은 아래와 같다.

1. 계획 단계

프로젝트 착수 시 보안 목표와 범위를 명확히 설정하고, 개인정보 보호나 규제 요건을 체크리스트로 준비한다. 예를 들어 "스마트 컨트랙트가 처리하는 데이터 중 민감정보는 어떤 것이 있으며, 이를 보호하기 위한 법적 요구사항을 충족시켰는가?" 와 같은 확인을 수행한다.

2. 요구사항 단계

기능 요구사항 보안 요구사항 명세서를 작성하고 위협 모델링을 실시한다. 예를 들어 "과거 유사 시스템에서 발생한 보안 사고 유형을 분석하여 본 시스템에 적용 가능한 보안 요구사항으로 반영하였는가?"와 같은 요소들을 점검한다.

3. 설계 단계

설계 검토 시 위협 모델을 바탕으로 보안 통제 항목이 설계에 포함되었는지 확인한다. 예를 들어 "권한 관리, 인증, 데이터 암호화, 입력 검증 등이 설계 문서에 반영되어 있는가?" 또는 "failure 없이 보안 아키텍처를 구성했는가?" 등을 확인한다.

4. 구현 단계

시큐어 코딩 체크리스트를 활용하여 코딩 단계에서 흔한 취약점이 발생하지 않도록 검사한다. 예를 들어 "산술 오버플로우/언더플로우를 방지하기 위해 SafeMath 라이브러리를 사용하였는가?", "외부 입력값에 대한 검증을 실시하고 있는가?", "에러 발생 시 revert/require를 적절히 사용하여 상태를 롤백하는가?" 등을 개발자가 자체 점검한다. 또한 코드 리뷰나 자동화된 정적 분석 도구를 통해 보안 약점의 유무를 확인한다.

5. 테스트 단계

개발 완료 후 보안 테스트 계획을 수립하고 정적/동적 분석 도구 및 퍼징 도구를 활용한다. 예를 들어 "슬리더나 Mythril 등의 도구로 알려진 취약점 패턴을 모두 점검하였는가?", "퍼징을 통해 reentrancy나 예외 처리 등 런타임 취약점을 탐지하였는가?" 등을 확인한다. 발견된 취약점은 재시험을 거쳐 모두 해결했는지 확인하고, 운영 단계에 돌입하기 전에 충분한 보안

검증이 이루어졌는지 평가한다.

이와 같이 각 단계 별로 보안 점검 활동을 한 후, 각 위협에 대한 완화 기법을 적용해야 하는데, SSDLC에서 정의하는 위협 완화는 다음과 같다.

A. 요구사항 정의(Requirements) 단계에서의 위협 완화

요구사항 정의 단계는 소프트웨어의 기능적 요구사항을 설정하는 것뿐만 아니라, 보안 요구사항도 명확히 정의하는 단계이다. 이 단계에서 위협 보안내재화를 수행하면, 소프트웨어가 어떤 보안 위협에 노출될 수 있는지를 미리 파악할 수 있다.

이때, 스마트 컨트랙트의 소프트웨어의 경우 소프트웨어 관점의 위협완화전략인 방법인 SWC Registry와 EEA Checklist와 같은 도구를 활용하여, 과거에 발생한 보안 사고를 분석하고, 유사한 위협을 식별할 수 있다. 또한 CWE나 CVE 같은 도구들 또한 소프트웨어 개발에 있어서 큰 도움을 제공할수 있다. 이러한 도구들은 기존의 보안 사고와 그 원인들을 체계적으로 정리한 자료로, 개발자들이 소프트웨어의 보안 요구사항을 설정할 때 참고할 수 있는 유용한 정보를 제공한다. 이를 통해, 소프트웨어가 직면할 수 있는 구체적인 보안 문제를 파악하고, 이를 해결하기 위한 보안 요구사항을 수립할 수 있다. 이 과정은 소프트웨어의 전반적인 보안 전략을 수립하는 데 매우 중요한 역할을 한다.

B. 설계 단계에서의 위협 완화

설계 단계에서는 요구사항 정의 단계에서 도출된 보안 요구사항을 바탕으로 소프트웨어 아키텍처를 설계한다. 이 과정에서 보안내재화는 식별된 위협 요소를 반영하여, 아키텍처 수준에서의 보안 대책을 마련하는 데 필수적이다. 예를 들어, 데이터 암호화, 권한 관리, 접근 제어와 같은 보안 메커니즘을 설계에 포함시켜, 식별된 보안 위협에 대응할 수 있는 아키텍처를 구축하는 것이 이 단계의 주요 목표이다. 초기의 보안내재화는 통해 잠재적인 보안 취약점을 미리 식별하고, 이를 반영한 설계를 통해 개발 후반부에 발견될 수 있는 보안 문제를 사전에 예방할 수 있다. 이렇게 하면, 개발이 진행되는 동안 발생할 수 있는 보안 문제를 최소화할 수 있으며, 궁극적으로 더 안전한 소프트웨어를 개발할 수 있다.

C. 테스트 단계에서의 위협 완화

테스트 단계는 소프트웨어 개발 과정에서 요구사항 정의 및 설계 단계에서 도출된 보안 요구사항과 보안 설계가 실제로 구현되었는지를 검증하는 중요한 역할을 한다. 이 단계에서는 소프트웨어의 기능적 요구사항뿐만 아니라, 보안 요구사항이 제대로 구현되었는지를 확인하기 위해 다양한 테스트 방법이 사용된다. 특히 스마트 계약 소프트웨어의 경우, 소프트웨어의 안전성을 확보하기 위해 정적 분석 도구와 동적 분석 도구를 활용하여 보안 취약점을 탐지하고 이를 해결하기 위한 조치를 취하는 것이 필수적이다.

테스트 단계에서의 보안 내재화는 다음과 같은 전략을 통해 수행될 수 있습니다.
첫째, 코드 리뷰와 정적 분석 도구를 사용하여 잠재적인 보안 취약점을 사전에 식별함으로써 배포 전에 코드의 안전성을 보장한다. 둘째, 동적 분석 도구를 활용하여 스마트 계약의 실행 중 발생할 수 있는 다양한 공격 벡터를 확인하고 대응한다. 셋째, 침투 테스트와 퍼징 테스트를 통해 시스템의 취약점을 식별하고, 보안 요구사항이 실제로 충족되었는지를 검증한다.

이 과정에서 EEA Checklist, SWC Registry, CWE와 같은 보안 표준을 참고하여 소프트웨어의 보안 수준을 확인할 수 있다. 마지막으로, 테스트 결과를 설계팀과 개발팀에 피드백하여 발견된 보안 취약점이나 문제점을 해결하고, 반복적인 테스트와 수정 작업을 통해 소프트웨어의 보안성을 지속적으로 향상시킨다.

이러한 테스트 단계에서의 위협 보안 내재화 활동은 소프트웨어의 최종 품질을 보장하고, 배포 이후 발생할 수 있는 보안 사고를 미연에 방지하는 중요한 요소이다. 이를 통해 개발된 스마트 계약 소프트웨어가 실제 운영 환경에서 안전하게 동작할 수 있도록 보장할 수 있다.

6. 소프트웨어 약점에 대한 정량적 보안평가 도구

소프트웨어 취약점에 대한 정량적 보안 평가는 시스템 보안성을 체계적으로 평가하고 강화하는 데 중요한 역할을 한다. 이 과정은 취약점을 수치화하여 그 심각성을 객관적으로 분석하고, 이를 바탕으로 우선순위를 정해 효율적인 보안 대응을 가능하게 한다. 정량적 보안 평가는 취약점의 위험도, 공격 가능성, 그리고 시스템에 미치는 영향 등을 고려하여 점수를 산출한다. 이러한 점수는 취약점이 시스템에 미치는 잠재적 위험을 평가하는 데 사용되며, 조직이 어떤 취약점에 먼저 대응할지를 결정하는 데 도움을 준다. 이 접근 방식은 주관적인 판단을 배제하고, 일관된 기준을 통해 취약점을 평가하는 장점이 있다. 대표적인 예로는 CVSS와 CWSS가 있으며, 이들은 각각 취약점의 심각성을 평가하고 관리하는 데 중요한 도구로 사용된다.

A. CWSS(Common Weakness Scoring System)

Common Weakness Scoring System (CWSS)는 보안 위험 평가에서 중요한 역할을 수행한다. CWSS는 소프트웨어 개발자들을 위해 만들어졌으며, 소프트웨어 개발자들은 코드의 약점을 지적하는 수많은 버그 보고서를 자주 접하게 되며, 그 중 일부는 악용 가능한 취약점으로 발전할 수 있다. 이러한 보고서의 방대한 양은 제한된 정보 속에서 어떤 문제를 우선적으로 처리해야 할지 결정하는 우선순위 설정 과정을 필요로 한다

CWSS는 Basic Finding, Attack Surface, Environmental이라는 세 가지 주요 메트릭 그룹으로 구성되어 있다. 각 그룹은 특정 약점에 대한 CWSS 점수를 산출하는 데 기여하는 다양한 메트릭 또는 요인들을 포함하고 있다. Basic Finding 메트릭 그룹은 약점이 내포하는 고유한 위험, 발견의 정확성에 대한 신뢰도, 그리고 적용된 통제의 견고성을 평가한다. Attack Surface 메트릭 그룹은 약점을 악용하기 위해 공격자가 직면하게 되는 장애물들을 평가한다. 마지막으로, Environmental 메트릭 그룹은 특정 환경이나 운영 맥락에서의 약점의 속성을 고려한다.

기본 발견 메트릭 그룹에서는 각 요인에 특정 값이 부여되며, 이 값은 기본 발견 하위 점수를 계산하기 위해 가중치로 변환된다. 이 하위 점수는 0에서 100 사이의 값을 가질 수 있다.

동일한 접근 방식이 공격 표면 및 환경 메트릭 그룹에도 적용되며, 이들의 하위 점수는 0에서 1 사이의 값을 갖는다. 결국 세 가지 하위 점수는 곱셈을 통해 최종 CWSS 점수를 산출하게 되며, 이 점수는 0에서 100 사이에 위치하게 된다.

B. CVSS(Common Vulnerability Scoring System)

Common Vulnerability Scoring System (CVSS)는 취약점을 일관되고 재현 가능한 방식으로 평가하고, 순위를 매기며, 보고하기 위한 표준화된 방법을 제공한다. CVSS의 주요 목적은 다양한 애플리케이션에서 발생하는 취약점을 공정하게 비교할 수 있도록 하는 것이다. CVSS는 각 취약점에 0에서 10까지의 점수를 부여하며, 0은 영향이 거의 없음을, 10은 가장 심각한 취약점을 나타낸다. 이러한 점수 체계는 취약점을 체계적으로 평가하고 전달함으로써, 이해관계자들이 대응의 우선순위를 효과적으로 정할 수 있도록 한다.

CVSS는 기본 메트릭 그룹, 일시적 메트릭 그룹, 환경 메트릭 그룹의 세 가지 메트릭 그룹으로 구성되어 있다. Basic Metric Metric그룹에는 취약점이 악용될 수 있는 용이성과 방법을 평가하는 악용 가능성 메트릭이 포함된다. Temperoral Metric은 시간이 지남에 따라 변할 수 있는, 예를 들어 패치나 악용 코드의 가용성 같은 취약점의 특성을 고려한다. Environmental은 해당 시스템의 특정 맥락, 즉 조직의 환경에서 취약점이 미칠 잠재적 영향을 평가한다. 이러한 세 가지 메트릭 그룹을 통합함으로써, CVSS는 취약점의 심각성과 잠재적 영향을 포괄적으로 평가할 수 있으며, 이를 통해 취약점 관리에서 더욱 신중하고 정보에 입각한 의사 결정을 내릴 수 있도록 지원한다.

C. CWSS 와 CVSS 차이

두 평가 체계인 CWSS와 CVSS는 적용 범위와 목적에서 차이가 있다. CWSS는 개발 단계의 소프트웨어 약점 평가에 초점을 두고, CVSS는 운영 단계의 발견된 취약점 평가에 주로 활용된다.

CWSS와 CVSS는 모두 보안 평가를 위한 정량적 점수 체계이지만, 적용 목적과 평가 대상,

사용 시점에서 명확한 차이를 가진다. CWSS는 소프트웨어 개발 단계에서의 보안약점 중심의 평가 체계로, 개발자가 코드 작성 중 발생할 수 있는 잠재적 보안 결함을 사전에 식별하고, 그 심각도와 우선순위를 결정하는 데 중점을 둔다. 반면 CVSS는 운영 중인 시스템에서 발견된 실제 보안 약점(Vulnerability)의 심각도를 평가하기 위한 체계로, 보안 관리자나 운영자 입장에서 대응 우선순위를 판단하는 데 사용된다.

CWSS의 평가 대상은 CWE로 분류된 소프트웨어 약점이며, 점수 범위는 0부터 100까지로 높은 점수일수록 심각한 약점을 의미한다. 이에 비해 CVSS는 CVE로 식별된 실제 취약점을 평가하며, 점수는 0.0부터 10.0까지로 부여되며 점수가 높을수록 심각도가 높음을 나타낸다.

평가 구성 측면에서 CWSS는 Base Finding, Attack Surface, Environmental의 세 가지 메트릭 그룹으로 구성되어 있으며, 각각 약점의 고유 위험도, 공격 가능성, 환경적 영향을 고려하여 점수를 산정한다. 점수는 곱셈 방식으로 종합된다. 반면 CVSS는 Base, Temporal, Environmental 메트릭으로 구성되며, 악용 가능성, 패치 여부, 환경 민감도 등의 요소를 가중합 방식으로 종합하여 점수를 계산한다.

적용 시점에서도 차이가 있다. CWSS는 스마트 컨트랙트와 같은 소프트웨어를 설계하고 구현하는 초기 단계에서 약점 제거를 목적으로 사용되며, 개발 중 정적 분석 도구, 시큐어 코딩 가이드와 함께 사용된다. CVSS는 시스템이 실제로 배포된 이후, 운영 단계에서 발생한 취약점에 대해 보안 공지, 패치 관리, 대응 우선순위 결정 등 실질적인 보안 대응에 활용된다. 아래 표는 두 체계의 목적, 점수 체계 및 활용 시점의 차이를 비교한 것이다.

표1. CWSS와 CVSS 비교

구분	CWSS	CVSS
주요 목적	개발자 관점에서 코드상의 약점을 정량 평가하여 개발 단계에서 취약점 예방 및 우선순위 결정	보안 관리자/운영자 관점에서 발견된 시스템 취약점의 심각도 평가 및 공통된 위험 전달
평가 대상	CWE로 식별되는 소프트웨어 약점	CVE로 공표된 보안 취약점
점수 범위	0 ~ 100	0.0 ~ 10.0
주요 평가 요소	Base Finding + Attack Surface + Environmental 메트릭 (곱셈 기반 종합). 약점의 고유 위험, 공격 난이도,	Base + Temporal + Environmental 메트릭 (가중합 종합). 취약점의 악용 난이도, 영향도, 시간 경과에 따른 요소, 환경적 중요도

	환경 영향 반영	반영
적용 시점	개발 초기 설계~구현 단계: 코드 작성 중 발견되는 잠재적 약점의 사전 제거에 활용	배포 후 운영 단계: 공개된 취약점(CVE)에 대한 보안 공지, 패치 우선순위 결정 등에 활용
활용 사례	SWC 등과 연계하여 개발팀이 위험도 높은 약점을 우선적으로 수정, 보안코드 리뷰에 활용	CVE 공지에 CVSS 점수 포함하여 각 조직이 취약점 대응 계획 수립, 침투테스트 결과 보고 등

7. 스마트컨트랙트에 대한 보안약점 분석 및 체크리스트

스마트 컨트랙트 소프트웨어에서는 소프트웨어 관점에서 보안내재화를 진행하였다. 스마트 컨트랙트 시스템 상에서, 개발자는 보안 취약점을 개발 초기 단계에서 내재화하는 것이 매우 효율적이므로, 이를 기반으로 작업을 진행하였다. 이에 따라, 스마트 컨트랙트 개발 과정에서 필요한 보안 약점을 조사하고, 해당 약점을 CWE(Common Weakness Enumeration)와 매핑하였다. 이후 CWE에 정의된 구성 요소에 따라 CWSS(Common Weakness Scoring System)를 활용하여 정량적으로 위험도를 산정하였다. 이러한 CWSS 점수를 바탕으로 취약점의 우선순위를 결정하고, 그에 따라 개발자들이 사용할 보안 체크리스트를 도출하였다. 이 과정은 보안 취약점의 체계적 관리를 가능하게 하여, 스마트 컨트랙트의 전반적인 보안성을 강화하는 데 기여하였다.

1.스마트 컨트랙트에서의 약점에 대한 CWSS Score

스마트 컨트랙트의 잠재적 약점을 분석하기 위해, 먼저 SWC 레지스트리를 활용하여 관련 약점 목록을 도출하였다. 이후, 이러한 약점을 CWE와 매핑 하여 더 깊이 있는 분석을 진행하였다. 이 매핑 작업을 통해 CWSS 점수를 산정할 수 있었다.

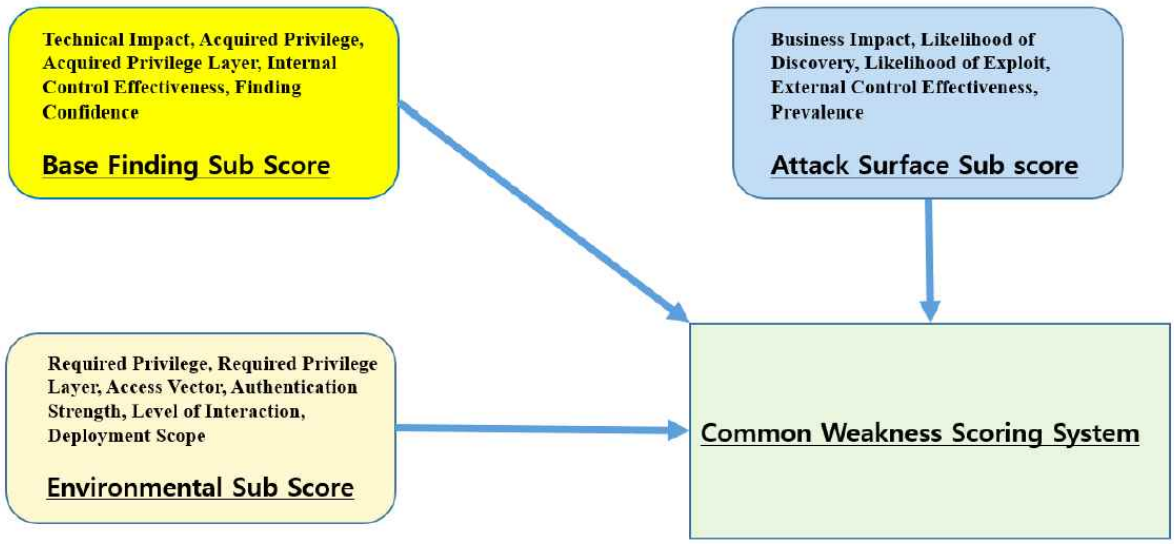

그림 3. CWSS 점수 산정방법

CWSS는 세 가지 주요 요소로 구성된다: Attack Surface Score, Environment Sub Score, 그리고 Basic Finding Score. 이러한 요소들은 각각 스마트 컨트랙트의 보안 취약성을 평가하는 데 중요한 역할을 하며, 각 항목의 가중치를 반영하여 총체적인 보안 점수를 산출한다. 이를 통해 스마트 컨트랙트의 보안 수준을 보다 정량적으로 평가하고 개선할 수 있는 기반을 마련할 수 있다.

CWSS는 크게 세 가지 하위 점수로 구성되어 있습니다. 첫 번째는 Base Finding Sub Score로, 발견된 약점의 기술적 영향을 평가한다. 여기에는 약점이 시스템에 미치는 기술적 영향, 공격자가 획득할 수 있는 권한, 권한이 적용되는 시스템 계층, 약점에 대한 내부 통제의 효과성, 그리고 약점 발견의 신뢰도 등이 포함한다.

두 번째는 Attack Surface Sub Score로, 약점이 얼마나 쉽게 발견되고 악용될 수 있는지를 평가한다. 이 점수는 약점이 비즈니스에 미치는 영향, 약점이 발견되거나 악용될 가능성, 외부 통제의 효과성, 그리고 약점이 얼마나 널리 퍼져 있는지 등을 고려한다.

세 번째는 Environmental Sub Score로, 약점이 실제 환경에서 얼마나 심각한지를 평가한다. 여기에는 필요한 권한, 접근 벡터, 인증 강도, 상호작용 수준, 그리고 약점이 배포된 범위 등이 포함된다. 이 세 가지 하위 점수가 결합되어 CWSS의 최종 점수를 산출하며, 이를 통해 약점의 심각도를 종합적으로 평가할 수 있다.

CWSS의 최종점수 공식은 다음과 같다.

CWSS Score= (Base Finding Score×Attack Surface Metric×Environmental Metric)

표 2. 스마트 컨트랙트 보안 약점과 CWSS 점수

Name of Bug	CWE Number	Name of Bug in CWE	CWSS Basic Finding	CWSS Attack Surface	CWSS Environmental Metric Group	CWSS Total Score
Functions without a specified function visibility type are public by default.	CWE 710	Improper Adherence to Coding Standards	63.0	0.900	0.693	39.293
Integer Overflow and Underflow.	CWE 682	Incorrect Calculation	63.0	0.895	0.693	39.075
Floating Pragma settings for compiler versions and flags.	CWE 664	Improper Control of a Resource Through its Lifetime	54.6	0.895	0.567	27.708
Unchecked call return value - The return value of a message call is not checked.	CWE 252	Unchecked Return Value	82.8	0.840	0.788	54.807
Missing or insufficient access controls, allowing Ether withdrawal.	CWE 284	Improper Access Control	84.6	0.690	0.819	47.808
Unprotected SELFDESTRUCT instruction.	CWE 284	Improper Access Control	88.2	0.690	0.819	49.843
Reentrancy attack allows malicious contract calls.	CWE 841	Improper Enforcement of Behavioral Workflow	63.0	0.900	0.828	46.948
State variable default visibility.	CWE 710	Improper Adherence to Coding Standards	54.6	0.895	0.621	30.346
Uninitialized storage pointers.	CWE 824	Access of Uninitialized Pointer	82.8	0.805	0.774	51.590
Assert Violation	CWE 670	Always-Incorrect Control Flow	81.0	0.820	0.693	46.029

		Implementation				
Use of Deprecated Solidity Functions	CWE 477	Use of Obsolete Function	54.6	0.805	0.693	30.459
Delegate call to Untrusted Callee	CWE 829	Inclusion of Functionality from Untrusted Control Sphere	81.0	0.895	0.828	60.03
DoS with Failed Call	CWE 703	Improper Check or Handling of Exceptional Conditions	51.8	0.82	0.4305	18.29
Transaction Order Dependence	CWE 362	Concurrent Execution using Shared Resource with Improper Synchronization ('Race Condition')	44.0	0.895	0.828	32.61
Authorization through tx.origin	CWE 477	Use of Obsolete Function	50.4	0.96	0.64	30.97
Block values as a proxy for time	CWE-829	Inclusion of Functionality from Untrusted Control Sphere	22.2	0.91	0.828	16.73
Signature Malleability	CWE-347	Improper Verification of Cryptographic Signature	81.0	0.895	0.42	30.45
Incorrect Constructor Name	CWE-665	Improper Initialization	54.6	0.91	0.64	31.80
Shadowing State Variables	CWE-710	Improper Adherence to Coding Standards	66.6	0.895	0.42	25.03

Weak Sources of Randomness from Chain Attributes	CWE-330	Use of Insufficiently Random Value	70.2	0.91	0.828	52.89
Missing Protection against Signature Replay Attacks	CWE-347	Improper Verification of Cryptographic Signature	81.0	0.895	0.64	46.40
Lack of Proper Signature Verification	CWE-345	Insufficient Verification of Data Authenticity	70.2	0.91	0.64	40.88
Requirement Violation	CWE-573	Insufficient Verification of Data Authenticity	51.8	0.895	0.42	19.47
Write to Arbitrary Storage Location	CWE-123	Write-what-where Condition	73.8	0.91	0.828	55.61
Incorrect Inheritance Order	CWE-696	Incorrect Behavior Order	51.8	0.895	0.64	29.67
Insufficient Gas Griefing	CWE-691	Insufficient Control Flow Management	70.2	0.895	0.42	26.39
Arbitrary Jump with Function Type Variable	CWE-695	Use of Low-Level Functionality	79.2	0.91	0.64	46.13
DoS With Block Gas Limit	CWE-400	Uncontrolled Resource Consumption	51.8	0.895	0.42	19.47
Typographical Error	CWE-480	Use of Incorrect Operator	18.6	0.895	0.828	13.78
Right-To-Left-Override control character (U+202E)	CWE-451	User Interface (UI) Misrepresentation of Critical Information	75.6	0.91	0.64	44.03
Presence of unused variables	CWE-1164	Irrelevant Code	20.4	0.91	0.42	7.80
Unexpected Ether balance	CWE-667	Improper Locking	86.4	0.91	0.828	65.10

Hash Collisions With Multiple Variable Length Arguments	CWE-294	Authentication Bypass by Capture-repla	81.0	0.895	0.4305	31.21
Message call with hardcoded gas amount	CWE-655	Improper Initialization	64.4	0.91	0.828	48.52
Code With No Effects	CWE-1164	Irrelevant Code	20.4	0.895	0.64	11.69

2. 스마트 컨트랙트 보안요구사항 EEA 체크리스트 매핑

스마트 컨트랙트의 보안을 강화하기 위해, CWSS 을 활용하여 약점에 대한 정량적인 점수를 산정하였다. 이후, EEA(Enterprise Ethereum Alliance)에서 발행한 스마트 컨트랙트 보안 체크리스트와 이러한 약점을 매핑하여 분석을 진행하였다.

EEA 체크리스트의 각 항목과 스마트 컨트랙트의 잠재적 약점을 비교하여, 각 항목이 얼마나 중요한지 "S", "M", "Q"로 구분된 Requirement 레벨에 따라 분류하였다. 여기서 "S"는 가장 중요한 보안 요구 사항을, "M"은 중간 정도의 중요도를, 그리고 "Q"는 상대적으로 낮은 중요도를 나타낸다. 이러한 매핑을 통해 스마트 컨트랙트가 보안 요구 사항을 충족시키고 있는지 EEA에서 제공하는 체크리스트를 통해 명확하게 확인할수 있었다.

표 3. 스마트 컨트랙트 보안 약점과 EEA CHECKLIST Requirement 매핑

Nmae of Bug	EEA Checklist Requirement	EEA Checklist Requirement Level
Unexpected Ether balance	Ether amount discrepancy	S
Write to Arbitrary Storage Location	Arbitrary Storage Write	S
Unchecked call return value	Unchecked Return Value from External Call	S

Uninitialized storage pointers	Uninitialized Storage Pointer	S
Unprotected SELFDESTRUCT instruction	Unprotected Selfdestruct	S
Message call with hardcoded gas amount	Fixed Gas Amount in Message Call	S
Missing or insufficient access controls	Insufficient Access Control	S
Arbitrary Jump with Function Type Variable	Arbitrary Jump Using Function Type	S
Assert Violation	Assert Statement Misuse	S
Delegate call to Untrusted Call	Delegatecall to Untrusted Contract	S
Reentrancy attack	Reentrancy Vulnerability	S
Floating Pragma settings for compiler versions	Floating Pragma	M
State variable default visibility	Default Visibility for State Variables	M
Transaction Order Dependence	Transaction Order Dependence (TOD)	M
Hash Collisions With Multiple Variable Length Arguments	Hash Collision	M
Authorization through tx.origin	Authorization Through tx.origin	M

Incorrect Constructor Name	Constructor Name Misuse	M
Presence of unused variables	Unused Variable	Q
Code With No Effects	Code Without Effects	Q
Typographical Error	Typographical Error	Q
Block values as a proxy for time	Block Values as Time Proxy	Q
DoS With Block Gas Limit	Denial of Service (DoS) with Block Gas Limit	Q
Insufficient Gas Griefing	Gas Limit Griefing	Q
Shadowing State Variables	State Variable Shadowing	Q

3.스마트 컨트랙트 보안요구사항 체크리스트

스마트 컨트랙트 소프트웨어에서는 보안 취약점을 체계적으로 분석하기 위해 개발 과정에서 발생할 수 있는 보안 약점들을 조사한 후, 이를 CWE(Common Weakness Enumeration)와 매핑하고, CWSS(Common Weakness Scoring System)를 활용하여 각 취약점의 위험도를 정량적으로 산정하였다. 이렇게 산정된 점수를 바탕으로 취약점의 우선순위를 결정하였다.

SecDevOps 접근 방식에서는 개발자가 보안 체크리스트를 활용하여 계획, 설계, 그리고 테스트 단계에서 보안 내재화가 제대로 이루어졌는지 확인하는 것이 중요하다. 스마트

컨트랙트의 특성상, 구현 후에 보안을 추가적으로 내재화하는 것은 비용적, 효율성 측면에서 불리할 수 있기 때문에, 설계 단계에서부터 보안을 내재화하는 것이 가장 효과적이다. 이러한 접근 방식을 통해, 스마트 컨트랙트의 보안성을 사전에 확보하고, 시스템의 안전성을 강화하였다.

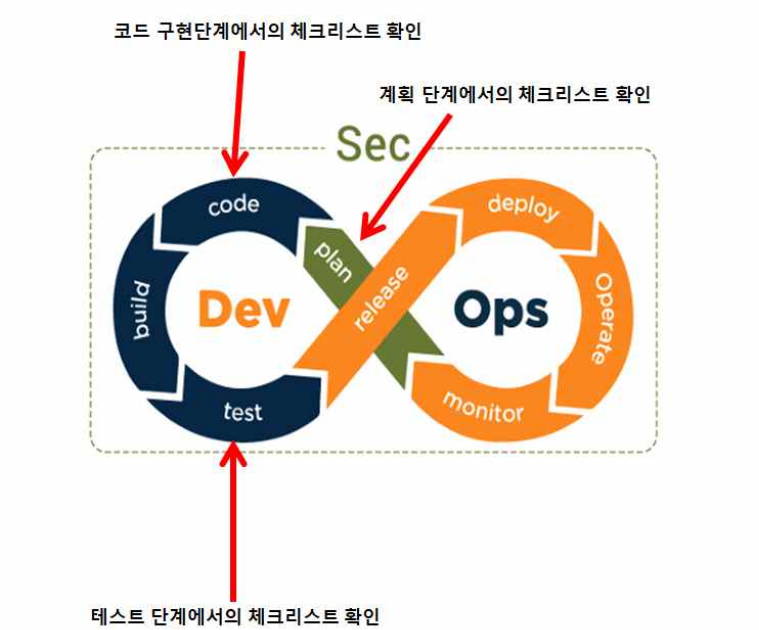

그림 4. DevSecOps에서의 보안체크리스트의 활용

따라서 이 프로세스를 지원하기 위해, 스마트 컨트랙트 개발의 계획, 설계, 그리고 테스트 단계에서 사용할 수 있는 체크리스트를 도출하였다. 이 체크리스트에는 각 단계에서 필요한 보안 요구사항이 명확히 제시되어 있으며, 이러한 요구사항을 검증할 수 있는 도구 또한 포함되어 있다. 개발자는 이 체크리스트를 통해 각 단계에서 요구되는 보안 내재화가 제대로 이루어졌는지를 확인하고, 필요한 보안 조치를 철저히 이행할 수 있다. 이를 통해 스마트 컨트랙트의 전반적인 보안성을 높이고, 개발 과정에서 발생할 수 있는 잠재적 위험을 효과적으로 관리할 수 있다.

스마트 컨트랙트의 보안 내재화를 위해서, 본 문서는 CWSS의 정량적 점수를 0바탕으로 체크리스트를 도출하였다. 점수가 높으면 반드시 그것에 대한 보안내재화는 이루어져야 하므로, 점수가 높은 순서대로 보안내재화의 Level 1부터 Level 3를 산정하였고, 점수가 높은 약점은 Level 1 Mandatory에 포함 시켰다. 또한 체크리스트의 Level 산정 방법은 다음과

같다.

Code 및 Test 단계에서는 스마트 컨트랙트가 보안 요구 사항을 충실히 반영하고 있는지 검증하는 것이 필수적이다. 이를 위해, 잠재적인 보안 약점을 식별할 수 있는 도구의 사용이 필요하다. 본 문서에서는 보안 체크리스트 항목과 이러한 보안 도구들 간의 매핑을 진행하여, 각 단계에서 효과적으로 검증을 수행할 수 있도록 안내한다.

표 4. 스마트 컨트랙트의 보안내재화를 위한 Level 산정

CWSS Total Score	Level
50 Above	Level 1 (필수항목)
30 - 49	Level 2 (추가조치)
30 Below	Level 3 (강화된 보안제어)

스마트 컨트랙트를 위한 보안요구사항 취약점 식별 도구는 아래와 같다.

A. VeriSmart : 스마트 계약의 보안을 검증하는 데 중점을 둔 정적 분석 도구이다. 이 도구는 Solidity로 작성된 스마트 계약에서 발생할 수 있는 다양한 보안 취약점을 식별하고, 계약이 예상대로 안전하게 작동하는지 확인한다.

보안 취약점 탐지 범위: integer over/underflow, division by zero, assertion violation, ether-leaking, suicidal, erc20 violation,reentrancy, tx.origin

B. ScFuzz :스마트 계약의 동적 분석을 중점으로 하는 도구이다. 이 도구는 계약이 실행될 때 발생할 수 있는 다양한 보안 취약점을 식별하고, 실시간으로 분석하여 계약이 예상대로 작동하는지 확인한다.

보안 취약점 탐지 범위: Assertion Failure, Arbitrary Write, Block State Dependency, Control-flow Hijack, Ether Leak, Freezing Ether, Integer Bug(over/underflow), Mishandled Exception, Multiple Send, Reentrancy, Requirement Violation, Suicidal Contract, Tranasaction Origin Use

C. LLM Based Tool (Large Language Model Based Tool): 정적분석 도구로 Solidity 언어로 작성된 스마트 계약에서 발생할 수 있는 주요 보안 취약점을 탐지하고 자동으로 수정하는 기술을 제공한다.

보안 취약점 탐지 범위: Arithmetic, Reentrancy, unchecked low level

D. Mythril: Mythril은 정적분석 도구로 Solidity 스마트 계약에서 잠재적인 보안 취약점을 탐지하기 위해 사용되는 오픈 소스 정적 분석 도구이다. 이 도구는 스마트 계약의 바이트코드를 분석하여 보안 결함을 발견하고, 다양한 공격 벡터를 탐지하는 데 중점을 둔다.

보안 취약점 탐지 범위: Delegate Call To Untrusted Contract, Dependence on Predictable Variables Ether, Thief Exceptions External Calls, Integer, Multiple Sends, Suicide State Change External Calls, Unchecked Retval,0 User Supplied assertion, Arbitrary Storage Write, Arbitrary Jump

E. Slither: Slither은 정적 분석 도구로 Solidity 스마트 계약에 특화된 정적 분석 프레임워크로, 보안 취약점 탐지와 코드 최적화를 위한 도구이다. 이 도구는 빠르고 유연하며, 계약 코드의 품질과 보안을 강화하는 데 도움을 준다.

보안 취약점 탐지 범위: Reentrancy, Unchecked Send, Unchecked Low-Level Calls, Integer Overflow/Underflow, Uninitialized Storage,Selfdestruct Instruction, Shadowing ,Unused Variables, Unrestricted Write to Storage, Authorization through tx.origin , Visibility Issues, Incorrect Constructor Names, State Variable Shadowing, Missing Return Statements, Constant Functions Modifying State, Unused Return Values, Low-Level Calls with Fixed Gas Amount, Block Information Dependency, Complex Fallback Functions, Dangerous Deprecated Opcodes, Uninitialized Storage Point

F. Manticore

Manticore는 스마트 계약과 바이너리 프로그램의 보안 분석을 위해 사용되는 도구이다. Manticore는 심볼릭 실행 기법을 통해 프로그램의 여러 실행 경로를 탐색하고, 각 경로에서 발생할 수 있는 오류나 보안 취약점을 감지한다.

이 과정에서 Manticore는 프로그램이 다양한 조건에서 어떻게 동작하는지를 분석하며, 예상치 못한 동작이나 잠재적인 문제를 식별한다. 특히 재진입 공격이나 잘못된 상태 변화와 같은 보안 취약점을 미리 발견할 수 있도록 한다. 이러한 기능은 프로그램이 실제 환경에서 어떻게 동작할지를 미리 시뮬레이션한다.

Manticore는 정적 분석 도구가 발견하기 어려운 런타임 문제를 탐지하는 데 특히 유용하며, 스마트 계약의 안정성과 보안을 강화하는 데 중요한 역할을 한다.

보안 취약점 탐지 범위: Delegate Call To Untrusted Contract, Dependence on Predictable Variables, Ether Thief, Exceptions in External Calls, Integer Overflows and Underflows, Multiple Sends, State Change After External Calls ,Unchecked Return Values (Unchecked Retval), User-Supplied Assertions, Arbitrary Storage Write, Arbitrary Jump

표 5. 스마트 컨트랙트 보안내재화를 위한 체크리스트

Checklist						
Weakness	Weakness Description	Plan Stage			Code Stage	Test Stage
		Level 1	Level 2	Level 3		
Arbitrary Jump with Function Type Variable	Potential vulnerability due to untrusted function pointer jumps.	✓			Slither	ScFuzz
Assert Violation	Error condition where an assertion fails, potentially leading to unintended behavior.	✓			VeriSmart	ScFuzz
Authorization through tx.origin	Insecure method of authorization using tx.origin, vulnerable to phishing attacks.		✓		VeriSmart	ScFuzz
Block values as a proxy for time	Using block values as a time proxy can be unreliable and manipulated			✓	NA	ScFuzz
Code With No Effects (Improper Coding Standards)	Code that does not affect the contract, indicating improper coding standards.			✓	Slither	NA
Delegate call to Untrusted	Risk of executing malicious code by	✓			VeriSmart, LLM Based	ScFuzz

Call	delegating calls to untrusted contracts.				Tool	
DoS With Block Gas Limit	Denial of Service (DoS) through excessive gas usage that exceeds block gas limits.			✓	NA	ScFuzz
Floating Pragma settings for compiler versions	Lack of fixed compiler version, leading to unexpected behavior in future compilations.		✓		Slither,	N/A
Hash Collisions With Multiple Variable Length Arguments	Vulnerability in hash functions with multiple variable-length arguments leading to collisions.		✓		Mythril	Manticore
Incorrect Constructor Name	Errors due to the incorrect naming of constructors in contracts.		✓		Slither	Mythx
Insufficient Gas Griefing	Transaction failures due to insufficient gas, potentially causing service disruptions.			✓	NA	ScFuzz
Message call with hardcoded gas amount	Fixed gas amounts in message calls, which can cause transaction failures.	✓			VeriSmart	Manticore
Missing or insufficient access controls	Lack of proper access controls, leading to unauthorized actions in the contract.	✓			VeriSmart	ScFuzz
Presence of unused variables	Variables that are declared but not used, indicating potential inefficiencies or mistakes.			✓	Slither	NA
Reentrancy attack	Vulnerability allowing repeated contract entry before previous	✓			VeriSmart, LLM Based Tool	ScFuzz

	execution completes.					
Shadowing State Variables	State variables being shadowed by local variables, causing unexpected behavior.			✓	Slither	ScFuzz
State variable default visibility (Improper Adherence Coding Standards)	Risk from state variables having default visibility that may not be intended.		✓		Slither	N/A
Transaction Order Dependence	Vulnerability where the order of transactions affects the outcome of a contract.		✓		Slither	ScFuzz
Typographical Error	Simple typo errors that can lead to unintended and critical issues in contract execution.			✓	NA	NA
Unchecked call return value	Failing to check return values of external calls, leading to missed errors.	✓			VeriSmart, LLM Based Tool,	ScFuzz
Unexpected Ether balance	Inconsistent or unexpected Ether balance, indicating potential errors or vulnerabilities.	✓			VeriSmart	ScFuzz
Uninitialized storage pointers	Storage pointers that are not initialized, leading to unexpected behavior or vulnerabilities.	✓			VeriSmart	Manticore
Unprotected SELFDESTRUCT instruction	Risk from allowing anyone to trigger the SELFDESTRUCT function, potentially destroying the contract.	✓			VeriSmart	ScFuzz
Write to Arbitrary Storage Location	Vulnerability allowing data to be written to arbitrary storage locations, possibly causing security	✓			VeriSmart	ScFuzz

	breaches.					

표 6. 약점 별 확인항목 체크리스트

Checklist			
Weakness	**적용 단계**	**점검 대상**	**확인 항목**
Arbitrary Jump with Function Type Variable	구현 단계	함수/ 어셈블리	함수 타입 변수에 대한 assembly 조작을 피하고, 외부 입력을 통해 함수 포인터 변수에 임의의 값이 할당되지 않도록 구현
Assert Violation	구현 단계	assert() 사용 로직	assert()가 실패할 수 있는 코드 경로가 없는지 확인하고 입력값 검증에는 require()를 사용하고, assert는 코드상의 불변조건 확인에만 사용 assert가 트리거될 수 있다면 해당 로직 버그를 수정하거나 require로 변경
Authorization through tx.origin	구현 단계	인증/권한 부여 로직	권한 체크에 tx.origin을 사용하고 있지 않은지 확인합니다. 인증에는 반드시 msg.sender를 사용하고, tx.origin == owner와 같은 패턴은 제거
Block values as a proxy for time	설계/구현 단계	블록 정보 이용 시각 로직	block.timestamp나 block.number를 정확한 시간 계산에 사용하지 않도록 설계
Code With No Effects (Improper Coding Standards)	구현/테스트 단계	데드 코드	실행 결과에 영향을 주지 않는 코드 확인 후 제거
Delegate call to Untrusted Call	설계/구현 단계	delegatecall 사용 부분	delegatecall을 사용할 경우 대상 주소가 신뢰 가능한지를 확인 사용자 입력으로 임의의 주소에 delegatecall하지 않도록 하고, 업그레이드 패턴 등에서 delegatecall 대상은 오직 소유자가 지정한 검증된 계약만 가능하도록 화이트리스트를 적용
DoS With Block Gas Limit	설계/구현 단계	대량 데이터 처리 로직	배열 전체를 순회하거나 무제한 증가하는 loop 등이 있는지 점검 크기가 가변적인 배열/맵에 대해 전체를 한 번에 처리하는 패턴을 피하고, 필요한 경우 작업을 여러 트랜잭션으로 분할하여 블록 가스 한도 내에서 처리되도록 설계
Floating Pragma settings for compiler versions	구현 단계	컴파일러 pragma 설정	pragma solidity ^x.y.z처럼 가변 버전 범위가 아닌, 테스트된 안전한 컴파일러 버전으로 고정

Hash Collisions With Multiple Variable Length Arguments	구현 단계	abi.encodePacked 해시 활용	abi.encodePacked()로 여러 가변 길이 인자를 해싱할 때 입력 배치에 따라 동일 해시가 발생하지 않도록 점검
Incorrect Constructor Name	구현 단계	생성자 함수 선언	Solidity 0.5.0 미만 버전에서는 계약명과 동일한 함수명이 생성자로 인식되므로, 철자가 정확히 일치하는지 확인 컴파일러 버전을 최신으로 업그레이드하고 constructor() 키워드를 사용하여 생성자를 명시적으로 선언
Insufficient Gas Griefing	설계/구현 단계	외부 호출 가스 처리 로직	다른 계약을 호출하는 릴레이어나 포워더 기능이 있다면, 호출시 충분한 가스를 제공하지 않아 발생하는 실패 상황을 고려하여 트랜잭션을 대신 실행해주는 계정을 신뢰할 수 있는 계정으로 제한하거나, 호출 전에 남은 가스량을 체크하여 부족할 경우 트랜잭션을 거절하는 등의 조치를 취함
Message call with hardcoded gas amount	구현 단계	Ether 전송/호출 (transfer/send)	address.transfer()나 address.send()와 같이 고정 가스를 사용하는 호출 지양 재진입 방지 패턴이나 ReentrancyGuard 등을 통해 보안을 유지
Missing or insufficient access controls	설계/구현 단계	함수 접근제어 (onlyOwner 등)	관리자만 호출해야 하는 중요 함수에 접근 제한자가 누락되거나 불충분한지 확인 모든 외부 공개 함수 중 민감한 상태 변경 함수에는 적절한 접근 제어자를 추가하고, 계약 배포자 주소(Owner) 설정 및 권한 관리를 정확히 구현
Presence of unused variables	구현/리뷰 단계	선언 후 미사용 변수	선언만 되고 사용되지 않는 상태 변수나 지역 변수가 존재하는지 확인 후 제거
Reentrancy attack	설계/구현 단계	외부 호출 및 상태 변경 순서	외부로 Ether를 보낸다거나 외부 함수를 호출하는 코드가 있을 경우, 먼저 상태를 변경한 후 호출하도록 구현 또한 재진입 락을 적용하거나 상태 변경 후 외부 호출을 수행함으로써 재진입 공격을 방어
Shadowing State Variables	구현/리뷰 단계	상속 및 변수 이름 선언	상속 구조에서 부모와 동일한 이름의 상태 변수를 자식 계약에서 재선언 하였는지 확인 이름이 겹치지 않도록 해야함
State variable default visibility	구현 단계	상태 변수 가시성 선언	모든 상태 변수에 public/private/internal 중 명

(Improper Adherence Coding Standards)			시적인 가시성 지정이 되어 있는지 확인
Transaction Order Dependence	설계 단계	트랜잭션 처리 순서에 영향 받는 로직	특정 트랜잭션이 먼저 실행되는지에 따라 결과가 달라질 수 있는지 점검
Typographical Error	구현/테스트 단계	산술/논리 연산 및 코드 리뷰	오타로 인한 오류가 없는지 코드 리뷰 및 정적 분석을 활용해 확인
Unchecked call return value	구현/테스트 단계	외부 함수 호출 반환값 처리	저수준 함수 호출의 반환값을 반드시 확인 성공 여부를 체크하여 실패 시 대비
Unexpected Ether balance	설계/구현 단계	Ether 잔고 사용 로직	의도치 않은 Ether가 들어와도 기능에 영향이 없도록 예치/인출 로직을 설계
Uninitialized storage pointers	구현 단계	스토리지 포인터 변수 사용	로컬에서 구조체나 배열 타입의 스토리지 참조 변수를 선언만 하고 즉시 할당하지 않는 코드가 있는지 확인
Unprotected SELFDESTRUCT instruction	구현 단계	selfdestruct 호출 부분	selfdestruct를 누구나 호출 가능한지 확인
Write to Arbitrary Storage Location	구현 단계	배열 길이 조작 등 스토리지 쓰기	배열이나 매핑 등 데이터 구조의 크기를 변경하거나 인덱스를 조작하는 로직에서, 하나의 구조체에 대한 쓰기가 다른 데이터의 저장 공간을 덮어쓰지 않는지 확인

8. 결론

스마트 컨트랙트 보안 내재화는 소프트웨어 개발 초기 단계부터 철저한 보안 검토와 관리를 통해 시스템의 안전성을 강화하는 필수적인 절차이다. 본 연구에서는 스마트 컨트랙트 개발 시 보안 내재화를 효과적으로 실현하기 위해 CWE(Common Weakness Enumeration)와 매핑된 약점들을 식별하고, CWSS(Common Weakness Scoring System)를 활용하여 이들 약점의 위험도를 정량적으로 평가했다. 이로써, 각 취약점의 우선순위를 산정하고, 이에 따라 개발자들이 보안 요구사항을 명확히 이해 할 수 있도록 체크리스트를 제공하였다.

SevDevOps 환경에서는 개발자들이 계획, 설계, 그리고 테스트 단계에서 체크리스트를 통해 보안 내재화가 제대로 이루어졌는지 확인하는 것이 중요하다. 이는 스마트 컨트랙트가 구현된 후에 보안을 강화하는 것이 비용과 효율성 측면에서 비효율적이기 때문이다. 따라서 설계 단계에서부터 보안을 내재화하는 것이 필수적이며, 이를 통해 보안 위험을 사전에 방지할 수 있다. 이러한 접근 방식은 스마트 컨트랙트의 신뢰성과 안전성을 높이며, 궁극적으로 블록체인 생태계 전체의 보안을 강화하는 데 기여한다.

이를 위해, 본 연구에서는 계획, 설계, 테스트 단계에서 사용할 수 있는 체크리스트를 도출하였고, 이 체크리스트는 각 단계에서 요구되는 보안 기준을 충족하는지 검토할 수 있도록 설계되었다. 체크리스트는 또한 사용자가 필요한 도구를 명확히 식별하여 보안 요구사항을 효과적으로 검증할 수 있도록 구성되었다. 이러한 체계적인 접근은 스마트 컨트랙트의 보안성을 확보하는 데 중요한 역할을 하며, 개발 과정 전반에서 지속적인 보안 강화를 실현할 수 있는 기반을 제공한다.

Reference

[1] L. Brent, A. Nash, and P. Szilagyi, "Ethical considerations and mitigations in smart contract deployment," Proceedings of the 2018 IEEE International Conference on Blockchain, 2018

[2] M. Bartoletti, S. Lande, and L. Pompianu, "SmarTEST: Automated Testing of Ethereum Smart Contracts," IEEE International Conference on Software Testing, Verification and Validation Workshops, 2018

[3] K. Atzei, M. Bartoletti, and T. Cimoli, "A Survey of Attacks on Ethereum Smart Contracts (SoK)," Proceedings of the 6th International Conference on Principles of Security and Trust (POST), 2017

[4] F. Zhang, D. Cecchetti, K. Croman, A. Juels, and E. Shi, "Town Crier: An authenticated data feed for smart contracts," Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016, pp. 270-282.

[5] N. Grech, M. Kong, V. B. F. Gomes, P. Eugster, and I. Dillig, "MadMax: Surviving Out-of-Gas Conditions in Ethereum Smart Contracts," Proceedings of the ACM on Programming Languages, vol. 2, no. OOPSLA, pp. 1-27, 2018.

[6] J. Krupp and C. Rossow, "TEEther: Gnawing at Ethereum to Automatically Exploit Smart Contracts," Proceedings of the 27th USENIX Security Symposium, 2018

[7] L. Luu, D.-H. Chu, H. Olickel, P. Saxena, and A. Hobor, "Making Smart Contracts Smarter," Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, 2016

[8] Y. Hirai, "Defining the Ethereum Virtual Machine for Interactive Theorem Provers," Proceedings of the 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, 2017

[9] A. Mavridou and A. Laszka, "Designing Secure Ethereum Smart Contracts: A Finite

State Machine Based Approach," Proceedings of the 22nd International Conference on Financial Cryptography and Data Security (FC), 2018

[10] S. Wang, Y. Yuan, X. Wang, J. Li, R. Qin, and F. Y. Wang, "An Overview of Smart Contract: Architecture, Applications, and Future Trends," Proceedings of the 2018 IEEE Intelligent Vehicles Symposium (IV), 2018