안전한 스마트 컨트랙트를 위한 보안 약점 진단 도구

Security Weakness Diagnostic Tool for Secure Smart Contract

김 병 현[1*]
Byung-hyun Kim

요 약

블록체인을 활용한 기술들이 발전함에 따라 스마트 컨트랙트의 사용 또한 증가하고 있다. 블록체인 기반 스마트 컨트랙트가 디지털 경제에 혁신을 가져올 것으로 기대하지만 이 기술이 안정적으로 활용되기 위해서는 해결해야 할 보안 문제가 많다. 따라서 이더리움 스마트 컨트랙트의 보안 약점 항목을 정의하고 진단하여 보안 위험을 예방해야 할 필요가 있다.

본 논문은 스마트 컨트랙트의 보안 약점을 컨트랙트 실행 전 사전에 분석하기 위해 솔리디티의 보안 약점 항목을 정의하였고, 이를 기반으로 보안 약점을 진단하는 이더리움 스마트 컨트랙트 진단 도구인 Haechi를 설계 및 구현하였다. 제안하는 진단 도구인 Haechi는 이더리움 스마트 컨트랙트를 작성하는 솔리디티 언어의 추상 구문 트리를 입력으로 받아 정의된 보안 약점을 사전에 분석하여 컨트랙트의 보안 위협을 제거한다.

☞ 주제어 : 보안 약점 진단, 정적 분석, 스마트 컨트랙트

ABSTRACT

The use of smart contracts is also increasing as the technology using blockchains develops. Although blockchain-based smart contracts are expected to revolutionize the digital economy, there are many security issues that need to be addressed before this technology can be used reliably. Therefore, it is necessary to define and diagnose security weakness items of Ethereum smart contracts to prevent security risks.

This paper defines the security weakness items of the chain code to analyze the security weaknesses of the smart contracts before the contract execution. Based on this, Haechi, an Ethereum smart contract diagnostic tool for diagnosing security weaknesses, is designed and implemented. The proposed diagnostic tool, Haechi, removes the contract's security threat by proactively analyzing the defined security weaknesses as an input of the abstract syntax tree of the solidity language that creates the Ethereum smart contract.

☞ keyword : Security Weakness Diagnosis, Static Analysis, Smart Contract

1. 서 론

블록체인을 활용한 기술들이 발전함에 따라 스마트 컨트랙트의 사용 또한 증가하고 있다. 블록체인 기반 스마트 컨트랙트가 디지털 경제에 혁신을 가져올 것으로 기대하지만 이 기술이 안정적으로 활용되기 위해서는 해결해야 할 보안 문제가 많다. 그 중 하나로 최근 이더리움에서 DAO, Parity 해킹 사건이 발생함에 따라 스마트 컨트랙트에 대한 신뢰가 떨어지고 있다[1-2]. 따라서 이더리움 스마트 컨트랙트의 보안 약점 항목을 정의하고 검사하여 보안 위험을 예방해야 할 필요가 있다.

본 논문에서는 스마트 컨트랙트의 보안 약점을 컨트랙트 실행 전 사전에 분석하기 위해 체인 코드의 보안 약점 항목을 정의하였고, 이를 기반으로 보안 약점을 진단하는 이더리움 스마트 컨트랙트 진단 도구인 Haechi를 설계 및 구현하였다. 제안하는 진단 도구는 정적 분석 방법을 통해 이더리움 스마트 컨트랙트 작성 언어인 솔리디티의 추상 구문 트리를 분석하여 진단한다. 이러한 진단 도구는 컨트랙트 실행 전 분석 하기 때문에 컨트랙트 실행 시

1 Department of Computer Engineering, Seokyeong University., Seoul, 02713, Korea.

☆ "이 논문은 2016년도 정부(미래창조과학부)의 재원으로 한국연구재단의 지원을 받아 실행된 기초연구사업임
(No.2016R1A2B4008392)"

필요한 비용이 들지 않으며 안전한 스마트 컨트랙트 실행을 보장할 수 있다.

2. 관련 연구

2.1 정적 프로그램 분석

정적 프로그램 분석 (Static program analysis)은 실제 실행 없이 프로그램을 분석하는 방법이다[3]. 대부분의 경우에 정적 분석은 소스 코드 형태로 분석하며 목적 코드 형태로 분석하기도 한다. 단순한 텍스트 비교나 문법적 구조를 비교하는 패턴 검사, 데이터의 흐름을 이용한 데이터 흐름 분석, 문법의 실행의미를 이용한 의미 분석이 있다.

정적 프로그램 분석은 프로그램을 실행시키지 않고 분석하기 때문에 프로그램을 실행하여 분석하는 방법인 동적 프로그램 분석에 비해서 짧은 시간에 넓은 범위를 분석할 수 있는 장점이 있다.

2.2 SECURIFY

(표 1) SECURIFY의 스마트 컨트랙트 보안 약점 항목

2018.06.	2019.06.
1. Transaction Reordering	1. Transaction Reordering
2. Recursive Calls	2. Recursive Calls
3. Insecure Coding Patterns	3. Insecure Coding Patterns
4. Unexpected Ether Flows	4. Unexpected Ether Flows
5. Use of Untrusted Inputs in Security Operations	5. Dependence on unsafe inputs

SECURIFY는 2017년 ETH Zurich의 블록체인 보안 프로젝트의 결과물로 이더리움 스마트 컨트랙트 진단 도구다[4].

SECURIFY는 정적, 동적 분석 등 자동화된 분석 기술을 통해 스마트 컨트랙트를 진단하고 그 결과를 사용자에게 전달하여 보안 위험을 예방한다. 표 1은 SECURIFY의 스마트 컨트랙트 보안 약점 항목을 나타낸다. 보안 약점 연구 진행으로 변경한 항목을 통해 최근 연구 방향과 변경하지 않은 항목에 대한 중요도를 알 수 있다. SECURIFY는 한 가지 항목 이름만 변경했고 대부분은 동일하다.

2.3 SmartCheck

SmartCheck는 SmartDec(모스크바의 소프트웨어 회사)에서 2018년에 개발한 소프트웨어로 스마트 컨트랙트 진단 도구다[5]. 표 2는 SmartCheck의 스마트 컨트랙트 보안 약점 항목을 나타낸다. SmartCheck는 두 가지 항목을 새롭게 변경하였다.

(표 2) SmartCheck의 스마트 컨트랙트 보안 약점 항목

2018.06.	2019.06.
1. No payable fallback function	1. No payable fallback function
2. Reentrancy	2. Compiler version not fixed
3. Unchecked math	3. Using tx.origin for authorization
4. Unchecked low-level call	4. Unchecked low-level call
5. Implicit visibility level	5. Implicit visibility level

3. 스마트 컨트랙트 보안 약점 항목 정의

본 논문은 이더리움에서 스마트 컨트랙트를 실행할 때 발생할 수 있는 보안 약점 항목을 정의하였다. 보안 약점 진단기에서 아래와 같은 항목을 기반으로 스마트 컨트랙트를 진단하고 그 결과를 사용자에게 전달하여 보안 위험을 예방할 수 있다. 표 3은 본 논문에서 제안하는 이더리움 스마트 컨트랙트 보안 약점 항목을 나타낸다.

(표 3) 제안하는 이더리움
스마트 컨트랙트 보안 약점 항목

항목	내용
1. 데이터 검증	1) 정수형 오버플로우 및 언더플로우
2. 재진입성	1) Ether 전송 함수
	2) Fallback 함수
3. 예외 발생	1) 서비스 거부
4. 컨트랙트 특성	1) 다중 상속
	2) 접근지정자
5. 특수 변수	1) tx 변수 사용

3.1 데이터 검증

검증되지 않은 데이터는 사용 시 보안 위험에 노출될 수 있기 때문에 이러한 보안 위험을 예방하기 위해서 검증된 유효한 데이터만 사용이 가능하도록 하여 보안 약점을 제거해야 한다.

3.1.1 정수형 오버플로우 및 언더플로우

오버플로우란 프로세스가 데이터를 저장할 때 최대 크기를 초과하는 데이터를 입력하면 정상적인 경우에는 사용되지 않아야 할 영역에 데이터가 덮어씌워 오류를 일으키거나 프로그램 보안 취약점을 야기 시키는 결함이다. 언더플로우는 최소 크기보다 작은 데이터를 입력하는 경우이다. 이 결함은 수십 년간 개발자를 괴롭혀온 오류이기 때문에 소프트웨어 보안에서 매우 중요하다. 솔리디티는 부호 없는 정수형 사용 시 최대 256비트 데이터를 처리할 수 있으므로 데이터 값을 증가시킬 때 2^{256} -1보다 클 경우 0값이 되고 감소시킬 때 0보다 작을 경우 2^{256}-1 값이 되어 올바르지 않은 값을 사용할 수 있다[6]. 그림 1은 산술 연산 시 안전하지 않은 코드의 예다.

```
1: Contract OverflowUnderflow {
2:     uint public zero = 0;
3:     uint public max = 2**256-1;
4:     function increment() public {
5:       max += 1;
6:     }
7:     function decrement() public {
8:       zero -= 1;
9:     }
10:}
```

(그림 1) 오버플로우 및 언더플로우 예제 코드

데이터를 사용하기 전에 예외 발생 함수 assert()를 통해 경계 검사를 함으로써 결함이 발생할 가능성이 존재하는 트랜잭션을 무효로 만들어 보안 약점을 제거할 수 있다. 그림 2는 산술 연산 시 안전한 코드의 예다.

```
1: Contract OverflowUnderflow {
2:     uint public zero = 0;
3:     uint public max = 2**256-1;
4:     function increment() public {
5:       assert(max < (2**256-1));
6:       max += 1;
7:     }
8:     function decrement() public {
9:       assert(zero > 0);
10:      zero -= 1;
11:    }
12:}
```

(그림 2) 산술 연산 시 안전한 예제 코드

3.2 재진입성

재진입성이란 함수가 여러 곳에서 동시에 호출되어도 문제가 발생하지 않는 함수의 성질을 말한다. 사용자가 계약에 송금한 Ether 잔액을 사용자별로 관리할 때, 함수를 이용해 잔액을 인출 한다면 문제가 발생할 수 있다. 재진입성 공격은 DAO 해킹 사건과 관련된 내용이며 원리를 이해하고 예방해야 한다.

3.2.1 Ether 전송 함수 재진입

이더를 전송하고자 하는 컨트랙트의 fallback 함수 내부에 외부 호출 코드가 작성되어 있을 경우 address.call.value().gas()(),address.send(), address.transfer() 각각의 특징을 고려하지 않고 사용하여 예상치 못한 외부 호출을 할 수 있다. address.call.value().gas()()는 fallback 함수의 명령어를 실행하기 위한 가스를 지정하기 때문에 악의적인 코드가 작성되어 있을 경우 예상하지 못한 코드를 실행할 수 있다. address.send()는 fallback 함수가 명령어실행을 위해 요구되는 가스량을 2300으로 제한한다. 이더 전송이 실패할 경우 false를 반환한다. 마지막으로 address.transfer() 또한 가스량을 2300으로 제한하지만, 이더 전송이 실패 시 예외 발생으로 트랜잭션을 무효화 시킨다.

fallback 함수의 명령어를 실행시키기 위해서는 address.call.value().gas()()를 통해 가스량을 설정해주고, fallback 함수가 전송 받은 이더만 처리할 수 있도록(2300 가스) 제한할 때 address.send()이나 address.transfer() 사용을 한다. 하지만 fallback 함수 내에서 외부 호출의 위험이 있기 때문에 전송 가스를 2300으로 제한하는 것을 지향한다. 이더 전송이 실패할 경우 address.transfer()는 예외를 발생으로 데이터 보호에 있어 address.send() 보다 안전하기 때문에 이더 전송 함수는 address.transfer()을 사용하도록 한다. 그림 3은 Ether 전송 시 안전하지 않은 코드의 예다.

```
1: contract Bank {
2:     uinttotalDeposit;
3:     mapping(address => uint) balance;
4:     ……
5:     function withdraw(uint amount) public payable {
6:       require(amount <= balance[msg.sender]);
7:       balance[msg.sender] -= amount;
8:       totalDeposit -= amount;
9:       msg.sender.call.value(amount).gas(3000)();
10:    }
11:}
```

(그림 3) Ether 전송 시 안전하지 않은 예제 코드

address.call.value().gas()() 사용시 사용할 수 있는 가스량을 설정해주기 때문에 fallback 함수에서 예상치 못한 외부 호출이 발생할 수 있다. 외부 호출은 그 계약이나 그 계약에 의존하는 다른 계약에 의해 악의적인 코드를 실행할 수 있으므로 잠재적인 보안 위험이다. 이를 예방하기 위해 공격자가 fallback 함수의 악의적인 코드를 삽입하고 계약을 배포하더라도, 이더를 전송하는 사용자가 address.transfer()를 사용하여 fallback 함수가 현재 전송 받은 이더만 처리할 수 있도록 한다면 예상하지 못한 보안 위험이 존재하더라도 그것을 예방할 수 있다[7]. 그림 4는 Ether 전송 시 안전한 코드의 예다.

```
1: contract Bank {
2:     uinttotalDeposit;
3:     mapping(address => uint) balance;
4:     ……
5:     function withdraw(uint amount) public payable {
6:       require(amount <= balance[msg.sender]);
7:       balance[msg.sender] -= amount;
8:       totalDeposit -= amount;
9:       msg.sender.transfer(amount);
10:    }
11:}
```

(그림 4) Ether 전송 시 안전한 예제 코드

3.2.2 Fallback 함수 재진입

fallback 함수는 호출하는 함수가 콘트랙트에 존재하지 않거나 이더를 전송 받았을 때 대체 기능으로 자동 호출되는 함수로서, fallback 함수를 구현할 때 외부 호출 코드를 삽입한다면 공격자에 의해 예상하지 못한 위험을 야기할 수 있다. 그림 5는 Fallback 함수의 기능으로 안전하지 않은 코드의 예다.

Mallory contract에 이더를 전송할 경우 fallback 함수가 자동 호출되고 내부에서 SimpleDAO 객체 dao의 withdraw()를 통해 외부 호출을 하게 된다. withdraw()에는 호출한 contract주소로 이더를 전송하여 다시 Mallory contract의 fallback 함수가 호출되는 재진입성의 위험이 발생하게 된다[8].

```
1: contract SimpleDAO {
2:     bool lockCredit;
3:     mapping (address => uint) public credit;
4:     function queryCredit(address to) public returns (uint) { return credit[to]; }
5:     function withdraw(uint amount) public {
6:       require(!lockCredit);
7:       lockCredit = true;
8:       if(credit[msg.sender] >= amount) {
9:         msg.sender.transfer(amount);
10:        credit[msg.sender] -= amount;
11:      }
12:      lockCredit = false;
13:    }
14: }
15:
16: contract Mallory {
17:    SimpleDAO public dao
18:    = SimpleDAO(0xca35b7d915458ef540ade6068dfe2f44e8fa733c);
19:    address owner;
20:    function Mallory() { owner = msg.sender; }
21:    function() public { dao.withdraw(dao.queryCredit(this)); }
22:}
```

(그림 5) Fallback 함수의 재진입을 예방한 예제 코드

프로그램 코드 작성 시 외부 호출 코드 삽입이 불가피할 경우, mutex를 이용할 수 있다. 그림 4에서 withdraw()함수는 lockCredit값이 true일 경우 require()를 통해 예외를 발생시켜 함수가 실행되지 않도록 하고 함수가 종료되기 직전에 lockCredit 값을 false로 설정한다. 만일 호출된 함수가 실행도중 재진입을 하게 될 경우 lockCredit값이 true이기 때문에 재진입을 막을 수 있다. 호출된 설정예외 발생 함수 require()으로 bool 변수 lockCredit 값을 확인하여 외부 호출로 인한 재진입성을 예방한다. 그림 6은 Fallback 함수의 재진입을 예방한 예제 코드다.

3.3 예외 발생

예외 발생은 잘못된 트랜잭션에 의한 데이터를 보호하는 데 있다. 하지만 공격자는 예외 발생 코드를 이용하여 계약이 본 기능을 하지 못하도록 할 수 있다.

3.3.1 서비스 거부

시스템을 악의적으로 공격해 해당 시스템의 자원을 부족하게 하여 원래 의도된 용도로 사용하지 못하게 하는 공격으로서, 예외를 계속 발생시켜 공격할 수 있다. 예외가 발생하게 되면 모든 트랜잭션이 무효가 된다. 만일 어떠한 목적을 위한 트랜잭션이 항상 무효가 된다면 원래 의도된 용도로 사용할 수 없게 된다. DoS공격에 취약한 부분을 별도의 트랜잭션으로 분리한다면 예방 할 수 있다. 그림 7은 서비스 거부 공격 예제 코드다.

```
1: contract Auction {
2:    address currentLeader;
3:    uinthighestBid;
4:
5:    function bid() payable external{
6:      require(msg.value > highestBid);
7:      if(currentLeader != 0){
8:        currentLeader.transfer(highestBid);
9:      }
10:     currentLeader = msg.sender;
11:     highestBid = msg.value;
12:   }
13:}
```

(그림 7) 서비스 거부 공격 예제 코드

최고입찰자가 변경되었을 때, 기존의 currentLeader의 주소로 이더를 환불시 transfer()가 실패하면(입찰 후 계정 삭제 등) 트랜잭션이 무효가 되어 입찰기능을 사용할 수 없게 된다. 이를 해결하기 위해 Mapping을 이용하여 refunds에 addres인덱스로 uint값을 저장한다. Bid()함수에서 입찰가가 현재 최고입찰가 보다 크지 않을 경우 require()를 통해 무효화 시킨다. 그렇지 않을 경우 if문을 통해 이전 최고입찰자에게 환불할 이더를 refunds에 저장하고 있다. 이 후 최고입찰자의 주소값을 currentLeader에 할당하고 최고입찰가도 변경한다. 최고입찰자가 변경되었을 경우 자동으로 이전 최고입찰자에게 이더를 환불하는 것이 아니라 bid()와는 다르게 withdrawRefund()를 만들어 환불 받을 수 있다. 그림 8은 서비스 거부 예방 예제 코드다.

```
1: contract Auction {
2:     address currentLeader;
3:     uinthighestBid;
4:     mapping(address => uint) refunds;
5:
6:     function bid() payable external{
7:         require(msg.value >highestBid);
8:         if(currentLeader != 0){
9:             refunds[currentLeader] += highestBid;
10:        }
11:        currentLeader = msg.sender;
12:        highestBid = msg.value;
13:    }
14:
15:    function withdrawRefund() external {
16:        uint refund = refunds[msg.sender];
17:        refunds[msg.sender] = 0;
18:        msg.sender.transfer(refund);
19:    }
20:}
```

(그림 8) 서비스 거부 예방 예제 코드

3.4 컨트랙트 특성

솔리디티는 객체지향 특성을 가지고 있다. C++, Java 등 객체지향 언어에서 class를 사용하는 것처럼 솔리디티는 contract 지정어를 통해 계약을 추상화시키며, 정보 은닉이 가능하다. 또한 상속을 통해 코드의 재사용이 가능하고 중복 및 재정의를 통한 다형성을 보장한다. 하지만 이러한 특성으로 인해 예상하지 못한 오류가 발생할 수 있다.

3.4.1 다중 상속

다중 상속이란 OOP 특징 중 하나이며, 어떤 클래스가 하나 이상의 상위 클래스로부터 여러 가지 행동이나 특징을 상속받을 수 있는 것을 말한다. 만일 다중 상속 관계에서 같은 이름의 멤버 함수가 있다면 모호성이 발생할 수 있다.

솔리디티는 다중 상속 관계에서 모호성이 발생할 경우 가장 최근에 상속받은 컨트랙트를 처리하게 된다. 이것은 의도하지 않은 결과일 수 있으며 자바에서는 모호성 문제로 인해 클래스의 다중 상속을 지원하지 않는다. 모호성을 해결하기 위해 다중 상속을 지양한다. 그림 9는 다중 상속 예제 코드다.

```
1: contract ownedContract {
2:     uint a;
3:     function get() public constant returns(uint) {
4:         return a;
5:     }
6: }
7:
8: contract BaseNo1 is ownedContract {
9:     function set() public {
10:        a = 1;
11:    }
12: }
13:
14: contract BaseNo2 is ownedContract {
15:    function set() public {
16:        a = 2;
17:    }
18: }
19:
20: contract Final is BaseNo1, BaseNo2{}
```

(그림 9) 다중 상속 예제 코드

Final 컨트랙트를 생성 후 set()을 호출하였을 때, C++에서는 컴파일오류가 발생하지만 솔리디티는 BaseNo1과 BaseNo2에서 재정의된 set() 중 BaseNo2의 set()을 호출하게 되어 get()을 호출할 경우 a의 값은 2가 된다. 만일 BaseNo1 컨트랙트에 중요한 코드가 들어가 있다면 문제

가 될 수 있다. 솔리디티에서 상속을 사용할 때 컴파일러가 오류로 인지하지 않기 때문에 의도하지 않은 오류를 막기 위해서 다중 상속은 지양 한다.

3.4.2 접근지정자

솔리디티는 기본적으로 컨트랙트 내부 변수의 디폴트 접근지정자는 internal이며, 함수는 public이다. 접근지정자를 default로 하거나 잘못 사용할 경우 예상하지 못한 외부 접근이 이루어질 수 있다.

접근지정자를 명시적으로 표기해주지 않으면 디폴트 접근지정자가 적용되며, 함수의 경우 public이기 때문에 누구나 접근할 수 있어 예상하지 못한 위험에 빠질 수 있다. 그렇기 때문에 프로그램 코드 작성 시 접근지정자를 명시적으로 표기해야 한다. 그림 10은 디폴트 접근지정자 사용 예제 코드다.

```
1: contract WalletLibrary {
2:     // ...
3:     function initWallet(address[] _owners, uint _required, uint _daylimit) {
4:         //...
5:         initMultiowned(_owners, _required);
6:     }
7:
8:     function initMultiowned(address[] _owners, uint _required) {
9:         //...
10:    }
11:}
```

(그림 10) 디폴트 접근지정자 사용 예제 코드

Parity Wallet 해킹 사건의 주요 원인으로서 WalletLibrary에서 initWallet()은 지갑의 주인만 호출할 수 있어야 하는데 접근지정자를 명시적으로 표기하지 않아 public으로 간주되어 누구나 지갑을 생성하고 주인을 바꿀 수 있다. 함수의 접근지정자가 public일 경우 누구나 접근가능하며 internal일 경우 같은 컨트랙트내에 함수만 접근이 가능하다. external로 접근지정자를 설정할 경우 컨트랙트 내의 함수가 아닌 다른 컨트랙트에서 호출이나 사용자 계정의 직접적인 트랜잭션으로 접근이 가능하다. 그림 11은 명시적 접근지정자 사용 예제 코드로써 외부에서 접근이 불가능하도록 하기 위해 internal을 사용하고 있다.

```
1: contract WalletLibrary {
2:     // ...
3:     function initWallet(address[] _owners, uint _required, uint _daylimit) internal{
4:         //...
5:         initMultiowned(_owners, _required);
6:     }
7:
8:     function initMultiowned(address[] _owners, uint _required) internal{
9:         //...
10:    }
11:}
```

(그림 11) 명시적 접근지정자 사용 예제 코드

3.5 특수 변수

솔리디티에서는 블록체인에 대한 정보를 제공하는 특수 변수가 전역 네임 스페이스에 항상 존재한다. 이러한 변수를 통해 필요한 데이터를 쉽게 얻을 수 있는 장점이 있지만 이에 따른 보안 약점도 존재한다.

3.5.1 tx 변수

```
1:  contract MyWallet {
2:      address owner;
3:
4:      function MyWallet() public {
5:          owner = msg.sender;
6:      }
7:
8:      function sendTo(address to, uint amount) public {
9:          require(tx.origin == owner); to.transfer(amount);
10:     }
11: }
12:
13: contract AttackingContract {
14:     address attacker;
15:
16:     function AttackingContract() public {
17:         attacker = msg.sender;
18:     }
19:
20:     function() public {
21:         MyWallet(msg.sender).sendTo(attacker, msg.value);
22:     }
22: }
23:
24:
```

(그림 12) tx.orign 사용 예제 코드

트랜잭션에 관한 데이터를 가지고 있는 tx를 이용하여 필요한 데이터를 쉽게 얻을 수 있다. tx.gasprice,

tx.origin등이 존재하며 gasprice는 거래의 가스 가격 값을 가지고 origin은 거래의 최초 생성자 주소 값을 가진다. 하지만 이러한 데이터를 이용한 악의적인 계약을 배포하여 예상하지 못한 위험을 일으킬 수 있다[9]. 그림 12는 tx.orign 사용 예제 코드다.

사용자가 지갑에서 이더를 AttackingContract로 전송 했을 때, fallback함수가 자동으로 실행되고 MyWallet().sendTo()를 호출하며 생성자 주소인 attacker(공격자 주소)와 현재 트랜잭션의 이더를 매개변수로 넘긴다. MyWallet에서 owner의 값은 사용자의 주소값이 되고 tx.origin값 또한 사용자의 주소값이 된다. 결국 tx.origin과 MyWallet의 owner가 일치하게 된다. 그렇기 때문에 require()호출에도 예외가 발생하지 않고 to에게 이더를 전송하게 된다. to는 attacker의 주소로서 사용자는 자신의 전송 이더를 잃어버릴 수 있다.

tx.origin을 이용하여 컨트랙트를 배포한 생성자와 사용자를 비교하려고 하였지만 컨트랙트가 컨트랙트를 생성할 경우 예상하지 못한 오류가 발생할 수 있다. 이러한 오류를 예방하기 위해서 tx.origin 사용을 지양하고 만일 원점 주소를 알아야 할 때는 처음 컨트랙트를 배포한 주소를 매개변수로 연결하여 해결할 수 있다. 그림 13은 안전한 원점 주소 사용 예제 코드다.

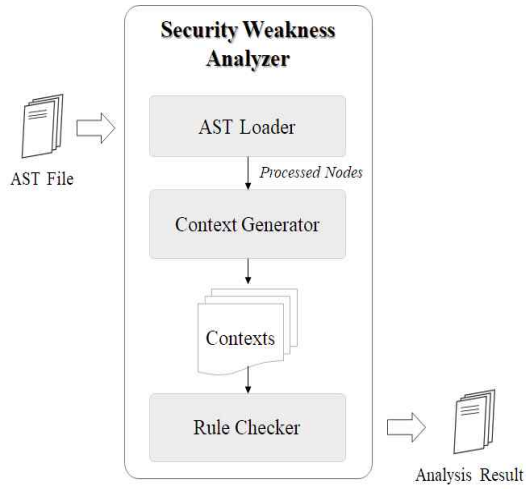
```
1: contract MyWallet {
2:     address owner;
3:
4:     function MyWallet() public {
5:         owner = msg.sender;
6:     }
7:
8:     function sendTo(address to, uint amount) public {
9:         require(to == owner); to.transfer(amount);
10:    }
11: }
12:
13: contract AttackingContract {
14:     address attacker;
15:
16:     function AttackingContract() public {
17:         attacker = msg.sender;
18:     }
19:
20:     function() public {
21:         MyWallet(msg.sender).sendTo(attacker, msg.value);
22:     }
22: }
23:
24:
```

(그림 13) 안전한 원점 주소 사용 예제 코드

4. 보안 약점 진단기

이 단락에서는 3절에서 정의한 보안 약점 항목을 정적 프로그램 분석 방법을 통해 진단한다.

정적 프로그램 분석에서 추상 구문 트리(AST : Abstract Syntax Tree)를 이용하면 불필요한 정보를 제거하고 분석하기 때문에 트리가 차지하는 메모리 공간의 낭비를 줄일 수 있고 진단 시간을 최소화 할 수 있다. 하지만 각 항목을 진단할 때마다 보안 약점 진단을 위한 정보를 얻기 위해 트리를 순회하는 것은 비효율적이다.



(그림 14) 보안약점 진단기 구조도

본 논문의 진단 도구는 솔리디티 컴파일러가 생성하는 추상 구문 트리를 이용하여 사전에 한 번의 트리 순회를 통해 진단에 필요한 데이터를 콘텍스트로 분류 및 저장한 후 사용한다. 이러한 방식은 보안 약점 진단 시간 비용을 줄일 수 있다. 그림 14는 제안하는 보안 약점 진단기의 구조도를 나타낸다.

4.1 AST 로더

솔리디티 컴파일러에 의해 생성된 추상 구문 트리는 Json 형식의 파일로 생성된다. 제안하는 분석기의 AST 로더는 솔리디티 컴파일러가 생성한 추상 구문 트리의 노드들의 필요한 정보들을 추려 사전에 정의된 노드 정보들로 가공된다. 가공된 노드들은 분석 비용을 줄이기 위한 콘텍스트 생성에 사용된다.

4.2 보안 약점 분석을 위한 콘텍스트 생성

보안 약점 항목을 진단할 때마다 트리를 순회하는 것은 비효율적이다. 따라서 각 보안 약점 진단에 필요한 유사한 속성을 가지는 데이터들을 사전에 분류 및 저장하여 사용해야 한다.

콘텍스트는 진단에 필요한 리소스 집합으로, 이를 이용하면 각 보안 약점 항목을 진단할 때 트리 순회를 하지 않아도 된다. 이러한 과정은 진단 시간을 줄여 프로그램의 성능을 향상시킬 수 있다. 표 4는 진단에 필요한 콘텍스트 항목이다.

(표 4) 보안 약점 분석에 사용되는 콘텍스트

항목	설명
1. Contract	컨트랙트 정의 노드 집합
2. Expression	표현식 노드 집합
3. Function Definition	함수 정의 노드 집합
4. Function Call	함수 호출 노드 집합
5. Variable	변수 선언 노드 집합

콘텍스트는 트리 노드 클래스에 선언된 정적 타입 리스트를 이용하여 생성한다. 리스트는 타입에 맞게 모든 노드 데이터를 가지고 있다. 콘텍스트는 이를 이용하여 필요한 데이터를 제공할 수 있도록 구성된다. 콘텍스트 항목에 대한 내용은 다음과 같다.

1. Contract 콘텍스트는 컨트랙트 데이터들의 집합이며 각 컨트랙트의 ID, 이름, 종류 등으로 구성된다. 이는 솔리디티 코드에서 컨트랙트의 종류나 개수를 파악하는 데 있어 유용하며 다중 상속이나 여부를 판단하는 데 사용된다.

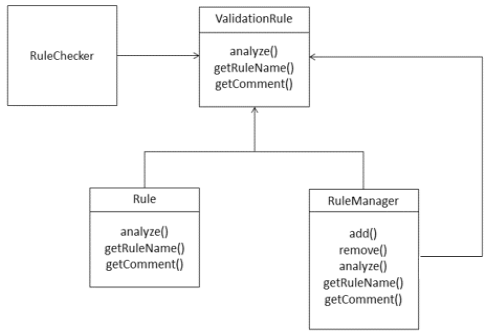
2. Expression 콘텍스트는 표현식과 관련된 데이터들의 집합이며 정수 오버플로우 등 연산과 관련된 진단에 사용된다.

3. FunctionDefinition 콘텍스트는 함수 정의와 관련된 데이터를 가지며 ID, 이름, 접근지정자 등이 있다. 이는 접근지정자 여부를 진단하는 데 사용된다.

4. FunctionCall 콘텍스트는 함수 호출과 관련된 데이터를 가지며 제어 흐름 그래프를 생성하는데 사용되고, FunctionDefinition 콘텍스트와 함께 재진입 여부를 진단하는 데 사용된다. 즉, 함수 호출 관계를 파악할 때 사용된다.

5. Variable 콘텍스트는 변수 선언과 관련된 데이터를 가지며 변수의 자료형, 상태, 저장 방식 등이 있다. 이는 특수 변수 사용 여부를 진단하는 데 사용할 수 있다.

4.3 규칙 검사기

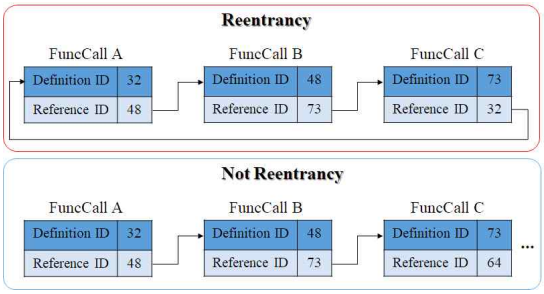


(그림 15) Haechi의 컴포지트 패턴 구조

규칙 검사기는 컴포지트 패턴[10]방식으로 동작한다. 진단을 위한 규칙 클래스들은 공통된 인터페이스를 상속받아 종류에 따라 재정의한다. 규칙 검사기는 정의된 규칙 개수만큼 인터페이스 객체를 생성하고 이를 이용해 보안 약점 항목을 진단한다. 그림 15는 본 논문의 컴포지트 패턴 구조도이다.

4.3.1 재진입성 검사

Fallback 함수를 악용한 재진입성 공격 예방을 위해서는 Ehter 전송 함수 호출 시 contract.call() 대신 transfer()를 사용해야 한다. Haechi는 Expression, FunctionCall 콘텍스트를 이용해 호출 하는 함수의 형태를 파악하여 진단한다. 사용자 정의 함수의 재진입성 검사의 경우에는 FunctionCall 콘텍스트에서 참조하는 함수의 id와 참조되는 함수의 id를 이용하여 진단할 수 있다. 그림 16은 추상 구문 트리의 함수 호출 정보를 이용한 재진입 진단 구조다.



(그림 16) Haechi의 재진입 진단

붉은색 박스의 경우 함수 호출이 사이클이 생기게 된다. 따라서 Haechi는 붉은 박스에 해당하는 함수 호출 구조를 가지는 경우 재진입이라고 검사한다.

4.3.2 예외 발생 검사

이더리움은 블록 가스 제한을 지정하며 블록에 포함된 모든 트랜잭션의 가스량은 임계 값을 초과 할 수 없다. 무한 반복이 가능한 패턴은 기능 실행 가스 비용이 한계를 초과 할 때 스마트 컨트랙트에서 서비스 거부 조건이 될 수 있다.

Haechi는 반복문에서 조건에 사용되는 값을 변수 값으로 사용하는 경우를 진단한다. 이는 Expression 콘텍스트에서 노드 타입이 For, While, DoWhile Statement와 일치한 데이터를 이용해 조건에 사용되는 값을 검사한다.

4.3.3 컨트랙트 특성 검사

솔리디티에서 다중 상속 시 C3 선형화(C3 Linearization)를 사용하여 기능의 모호성을 해결한다[11]. 컨트랙트의 우선 순위에 따라 기능이 달라질 수 있으므로 상속 순서가 매우 중요하다. 다중 상속의 모호성은 Contract, FunctionDefinition 콘텍스트에서 컨트랙트 상속 구조와 함수의 형태를 비교하여 부모 컨트랙트 중 같은 형태의 함수가 존재하는지 검사한다.

컨트랙트안에 존재하는 함수는 접근지정자를 작성하지 않을 경우 public으로 설정된다. Parity 해킹 사건과 관련된 내용이며 이러한 특성은 예상하지 못한 기능 실행을 불러올 수 있다. Haechi는 FunctionDefinition 콘텍스트를 이용하여 각 함수가 접근지정자를 사용했는지 검사한다.

4.3.4 특수 변수 검사

tx.origin은 거래를 보낸 계정의 주소를 반환하는 전역 변수로 정의되며 권한 부여에 tx.origin을 사용할 때, 권한을 가지는 악의적인 컨트랙트를 호출 할 경우 컨트랙트가 취약할 수 있다. Haechi는 Expression 콘텍스트 정보를 이용하여 tx.origin변수를 사용하고 있는지

검사한다.

5. 실험 결과

본 논문의 Haechi는 Smart Contract Weakness Classification and Test Cases에 정의된 보안 약점 중 SWC-107: Reentrancy, SWC-115 : Authorization through tx.origin, SWC-125: Incorrect Inheritance Order를 검사한다. 표 5는 SWC-107 예제 소스 코드와 진단 결과이다.

(표 5) SWC-107 예제 소스 코드 및 진단 결과

SimpleDao.sol

3: contract SimpleDAO {
4: mapping (address => uint) public credit;
5:
6: function withdraw(uint amount) public {
7: if (credit[msg.sender]>= amount) {
8: require(msg.sender.call.value(amount)());
9: credit[msg.sender]-=amount;
10: }
11: }
12: ... 중략 ...
13:}

Result
========== Reentrancy ========== [MAJOR] Reentrancy warning "Incorrect function usage in Ether transmission, Use transfer() instead", Line : 8

표5의 소스코드에서 Ether 전송 시 transfer() 함수가 아닌 contract.call() 함수를 호출 하고 있기 때문에 재진입 위험이 존재한다.

표 6은 SWC-115 예제 소스 코드와 진단 결과이다. 컨트랙트의 권한을 부여할 때, 그 기준을 tx.origin을 사용하고 있다.

(표 6) SWC-115 예제 소스 코드 및 진단 결과

mycontract.sol

3: contract MyContract {
4: address owner;
5:
6: function MyContract() public {
7: owner = msg.sender;
8: }
9:
10: function sendTo(address receiver, uint amount) public {
11: require(tx.origin == owner);
12: receiver.transfer(amount);
13: }

Result
========== tx-origin ========== [MAJOR] tx-origin warning "Potential vulnerability to tx.origin attack", Line : 11

(표 7) SWC-125 예제 소스코드 및 진단결과

MDTCrowdsale.sol

: ... 중략 ...
468: contract CappedCS is Crowdsale {
: ... 중략 ...
486: function validPurchase() internal constant returns (bool) {
: ... 중략 ...
489: }
490: }
491:
492: contract WhitelistedCS is Crowdsale {
: ... 중략 ...
504: function validPurchase() internal constant returns (bool) {
: ... 중략 ...
506: }
507: }
508:
509: contract MDTCrowdsale is CappedCS, WhitelistedCS {
: ... 중략 ...
519: }

Result
======== Multiple-Inheritance ======== [MAJOR] Multiple-Inheritance warning "Do not write multiple inheritance", Line : 509

표 7은 SWC-125 예제 MDTCrowdsale.sol 소스 코

드와 진단 결과이다. MDTCrowdsale 컨트랙트가 상속 받은 CappedCrowdsale, WhitelistedCrowdsale 컨트랙트에는 동일한 함수 validPurchase()가 존재하여 다중 상속 문제가 발생할 위험이 있다.

참고문헌(Reference)

[1] Quinn Dupont, “Experiments in Algorithmic Governance: A history and ethnography of ‘The DAO’, a failed Decentralized Autonomous Organization”, 2016.

[2] SANTIAGO PALLADINO, “The Parity Wallet Hack Explained”, https://blog.zeppelin.solutions/on-the-parity-wallet-multisig-hack-405a8c12e8f7, 2017.

[3] Anders Møller and Michael I. Schwartzbach, “Static Program Analysis”, pp.1-2, 2019.

[4] “Securify”, https://github.com/eth-sri/securify.

[5] “SmartCheck”, https://github.com/smartdec/smartcheck.

[6] Alber Erre, “Understanding uint Overflows and Underflows – Solidity (Ethereum)”, https://medium.com/3-min-blockchain/understanding-uint-overflows-and-underflows-solidity-ethereum-8603339259e6, 2018.

[7] Thomas Wiesner, “Solidity - .send(...) vs. .transfer(...)”, https://vomtom.at/solidity-send-vs-transfer, 2017.

[8] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli, “A survey of attacks on Ethereum smart contracts”, pp.13-14.

[9] “Solidity Recommendations : Avoid using tx.origin”, http://consensys.github.io/smart-contract-best-practices/recommendations/#avoid-using-txorigin.

[10] Dr. Michael Eichberg, “The Composite Design Pattern”, p.9.

[11] “Solidity 0.5.13 documentation : Multiple Inheritance and Linearization”, https://solidity.readthedocs.io/en/develop/contracts.html#multiple-inheritance-and-linearization.

[12] “What is Gas?”, https://kb.myetherwallet.com/gas/what-is-gas-ethereum.html.

[13] Mahendar B, “Secure Smart Contract Security in Solidity”, https://www.nvestlabs.com/2019/03/15/secure-smart-contract-security-in-solidity, 2019.

[14] Christian Reitwiessner, “Smart Contract Security”, https://blog.ethereum.org/2016/06/10/smart-contract-security, 2016.

[15] “Contract security techniques and tips”, https://github.com/ethereum/wiki/wiki/Safety, 2018.

[16] Nikhil Mohan, “Solidity Smart contract Security best practices”, https://lightrains.com/blogs/smart-contract-best-practices-solidity, 2017.

[17] Daniel Perez, Benjamin Livshits, “Smart Contract Vulnerabilities: Does Anyone Care?”, arXiv:1902.06701v1, 2019.

[18] “Known Attacks”, Ethereum Smart Contract Best Practices, https://consensys.github.io/smart-contract-best-practices/known_attacks.

[19] “Solidity Recommendations”, Ethereum Smart Contract Best Practices, https://consensys.github.io/smart-contract-best-practices/recommendations.

◐ 저 자 소 개 ◑

김 병 현(Byung-hyun Kim)

사진

2014년~현재 서경대학교 컴퓨터공학과 재학

관심분야 : 컴파일러, 가상기계.

E-mail : byunghyun@skuniv.ac.kr