**이더리움 스마트 컨트렉트에 대한 공격들의 조사**

Abstract. 스마트 컨트렉트들은 외부의 신뢰있는 권력의 필요 없이 분산화 된 노드들의 네트워크에 의해 정확하게 실행 될 수 있는 컴퓨터 프로그램들입니다. 스마트 컨트렉트들은 많은 양의 자산들을 다루고, 보내기 때문에, 컨트렉트들의 구현이 자산들을 훔치거나 손대는 것에 초점이 맞춰져있는 공격들에 대비하여 secure해야 한다는 것은 중요합니다. 우리는 가장 잘 알려져있는 이더리움에서 이 문제를 연구했고 스마트 컨트렉트들에 대한 프레임워크를 사용합니다. 우리는 이더리움 스마트 컨트렉트들의 보안 취약점들을 분석했습니다. 그리고 취약점들로 이어질 수 있는 공통적인 프로그래밍 결함의 분류를 제공할 것입니다. 우리는 상대방이 돈을 훔치거나 다른 문제를 야기할 수 있도록하는 이러한 취약점들을 익스플로잇한 공격들을 보여줄 것입니다.

**1 Introduction**

2009년에 런칭을 한 이후로 10 Billion 달러의 축적에 도달한 분산화된 암호 화폐인 비트코인의 성공은 산업과 학회에 엄청난 흥미를 불러일으켰습니다. 산업들에서 - 뿐만 아니라 중앙정부들도- 암호화폐들의 기술에 깔려진 블록체인의 “파괴적” 잠재력에 의해 매료되었습니다. 기본적으로, 블록체인은 p2p 네트워크의 노드들에 의해 유지되는 추가전용 데이터 구조입니다. 암호화폐들은 돈의 이중지불을 피하기위해 화폐의 모든 전송들을 기록하는 공공장부로서 블록체인을 사용합니다.

비록 비트코인은 블록체인 기술들의 가장 전형적인 어플리케이션이지만, 암호화폐를 넘어선 많은 어플리케이션들이 있습니다: e.g, 금융의 상품들과 서비스들, 다양한 종류의 자산들의 소유권 추적, 디지털 신원 조회, 투표, 등. 핫 토픽은 스마트 컨트렉트들을 구현하기 위해 어떻게 블록체인 기술들을 사용할 것인가에 대한 방법론입니다. 매우 추상적으로, 스마트 컨트렉트들은 신뢰있는 권력에 대한 의존 없이 블록체인의 합의 메커니즘에 의해 자동으로 강요되었지만, 서로를 불신하는 참가자들 사이의 합의들입니다.

스마트 컨트렉트들에 대한 가장 두드러진 프레임워크는 이더리움이다. 이더리움의 저축은 2015 7월 런칭 이후로 1 billion 달러에 도달했습니다 이더리움에서 스마트 컨트렉트들은 완전튜링 언어로 쓰여진 컴퓨터 프로그램들로 만들어졌습니다. 어떻게 p2p 네트워크의 노드들이 블록체인을 연장하는지 명시한 이더리움 합의 프로토콜은 컨트렉트들의 정확한 실행을 보장하는 목표를 가지고 있습니다. 그 프토토콜의 핵심 insight들중 하나는 블록체인에 새로운 블록을 추가하기 위해 노드들이 상금을 받을 확률이 그 노드의 계산의 파워에 비례하는 로또에 참여를 해야만합니다. 인센티브 메커니즘은 비록 로또에 당첨된 악의적인 노드가 부정확한 컨트렉트 실행들을 가지고 블록을 추가하려고 시도하더라도, 이 블록은 결국에는 블록체인에서 제거가 될 것입니다. 비록 합의 프로토콜의 효율성에 대한 비판이 있지만, 최근 이론적인 연구들은 정직한 노드들이 그 네트워크의 계산 파워의 대부분을 제어할 때 마다 합의 프로토콜의 보안을 수립하고 있습니다. 스마트 컨트렉트들이 정확하게 실행되는 이더리움이 그들의 효율성을 위해 조건을 거는 것은 필수적입니다. 다르게 말하면, 상대방이 합법적인 참가자들로부터 자신에게 약간의 돈을 가져가기 위해 실행을 조작할 수 있다는 말입니다. 그러나 실행 그 자체의 정확성은 스마트 컨트렉트들을 secure하게 만드는데 충분하지 않습니다. 이더리움 스마트 컨트렉트들에서의 여러 보안 취약점들은 개발 경험이나 이더리움 블록체인 기반의 모든 컨트렉트들의 정적 분석에 의해 발견이 됩니다. 이러한 취약점들은 돈의 손실을 불러온 이더리움 컨트렉트들에 대한 몇몇 실제 공격들에 의해 Explit됩니다. 이러한 공격들 중 가장 성공적인 공격은 한 컨트렉트로부터 60M 달러를 훔친 공격입니다. 하지만 공격의 효과는 블록체인의 논의된 변경 이후로 없어져버렸습니다.

이더리움에서 에러들을 일으키는 스마트 컨트렉트의 구현을 하게되는 이유는 여러가지가 있습니다. 그것들 중 많은 부분은 이더리움 기반의 high-level 프로그래밍 언어인 Solidity와 관련이 있습니다. 많은 취약점들은 Solidity의 Semantic들과 프로그래머들의 직관 사이의 조정 실패로 인해 발생되는 것으로 보입니다. 그러한 프로그램은 Solidity가 이상한 방식에 있는 특징들 중 몇몇 특징을 구현한 것입니다. 동시에, Solidity는 갑작스래 재요청을 하거나 딜레이가된 시점에서 계산 단계들이 public blockchain에 기록이 된다는 사실과 같은 특정 도메인 양상의 구성을 소개하지 않습니다.

또 다른 불안정한 스마트 컨트렉트들의 확산의 주된 원인은 알려진 취약점들의 문서가 여러 소스들을 통해서 공식 문서, 리서치 문서, 인터넷 포럼들을 포함해 퍼지고 있는 것입니다. 그리고 업데이트 된 이더리움 스마트 컨트렉트들에 대한 취약점들과 공격들의 Survey는 여전히 부족합니다.

**Contributions.** 이 paper에서 우리는 이더리움과 high-level programming 언어의 보안 취약점들에 대한 체계적인 설명을 제공할 것입니다. 우리는 분류에서 취약점들의 원인들을 조직화시켰습니다. 그것의 목적은 크게 두 가지입니다. (i) 스마트 컨트렉트들의 개발자들이 공통적인 결함들을 알고 피하기위한 레퍼런스를 만들기 위해 (ii) 연구자들에 대한 가이드로로서, 스마트 컨트렉트들에 대한 분석과 검사 기술의 개발을 장려하기 위해. 분류에서 취약점들의 대부분의 원인들에서, 우리는 그것들을 exploit하는 실제의 공격을 보여줄 것입니다. 우리의 모든 공격은 이더리움 testnet에서 테스트되었습니다. 그리고 그들의 코드는 co2.unica.it/ethereum에서 확인 가능합니다.

**2. 스마트 컨트렉트들에 대한 배경**

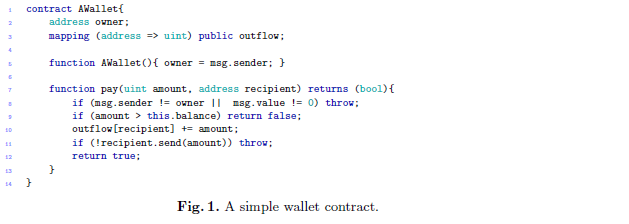
이더리움은 컨트렉트라고 불리우는 프로그램들을 가동하는 분산화된 가상 머신입니다. 컨트렉트들은 EVM 바이트코드라고 불려지는 완전 튜링 바이트코드 언어로 쓰여집니다. 한 컨트렉트는 각각 바이트 코드 명령들에 의해 정의된 함수들로 이루어져있습니다. 컨트렉트들의 눈에 띄는 특징은 그들이 ETHER를 유저 사이에서 그리고 다른 컨트렉트들에게 보낼 수 있다는 것입니다.

유저들은 이더리움 네트워크에 트랜잭션들을 보냅니다. (i) 새로운 컨트렉트들을 만들기 위해서 (ii) 컨트렉트의 함수들을 사용하기 위해서 (iii) ether를 컨트렉트들 혹은 다른 유저들에게 보내기 위해서. 모든 트랜잭션들은 블록체인이라고 불리우는 첨부전용 public 데이터 구조에 기록이 됩니다. 블록체인 위에서 트랜잭션들의 연속성은 각 컨트렉트의 상태와 각 유저의 balance를 결정합니다.

컨트렉트들이 경제적 가치를 가지기 때문에, 그들의 실행이 정확하게 되는 것을 보장하는 건 매우 중요합니다. 이 목적에 대해, 이더리움은 신뢰성 있는 중앙 권력에 대한 의존을 하지않습니다: 오히려, 각 트랜잭션은 마이너들이라고 불리는 불신하는 peer들의 큰 네트워크에 의해 작동되어집니다. 컨트렉트들의 실행에서 가능성 있는 논쟁들(공격들이나 실패들)은 “작업증명”에 기반한 합의 프로토콜을 통해 해결됩니다. 상대방이 그 네트워크의 계산 파워의 대부분을 제어할 수 없는 한, 컨트렉트들의 실행은 정확합니다.

합의 프로토콜의 보안은 정직한 마이너들이 합리적이라는 가정과 마이너들이 프로토콜을 공격을 시도하는 것보다 따르는 것이 더 편하다는 가정에 달려있습니다. 이 가정을 유지하기 위해, 마이너들은 그 프로토콜에 의해 요구되는 계산들을 하는 것에 대한 경제적 인센티브들을 받습니다. 이러한 요금들은 유저들이 시간-소비 computation들을 가진 네트워크를 장악하려고 시도하는 denial-of-service 공격들로부터 막는 트랜잭션의 실행에 관계가 있습니다.

**스마트 컨트렉트들의 프로그래밍.** 우리는 소유자에 관련된 개인의 지갑을 구현하는 작은 예시를 통해 컨트렉트들을 설명합니다. 곧바로 EVM bytecode로 프로그래밍하는 것보다는 오히려 우리는 EVM 바이크코드로 컴파일된 자바스크립트 같은 프로그래밍 언어인 Solidity를 사용한다. 그 컨트렉트는 다른 유저들로부터 ether를 받을 수 있고 그것의 owner는 pay 함수를 통해 다른 유저들에게 ethere를 보내줄 수 있다. 해쉬테이블 outflow는 돈을 보낸 주소들과 보낸 전체 액수를 기록한다. 그리고 모든 받은 ether는 컨트렉트에 의해 가지고 있게된다. 그것의 양은 자동으로 balance로 기록이 된다: 이것은 특별한 변수로, 프로그래머에 의해 바꿔질 수 없다.



컨트렉트들은 필드들과 함수들로 구성되어있습니다. 한 유저는 이더리움 노드들에 적절한 트랜잭션을 보냄으로써, 함수를 사용합니다. 그 트랜잭션은 실행비용( 마이너들을 위한 )을 포함하고 caller에서 컨트렉트로 ether의 전송을 포함합니다. Solidity는 특정 행동에서 예외 특징들이 있습니다. 예외가 throw되었을 때, 손을 댈 수가 없습니다: 실행이 멈추고, 실행비는 잃고, ether의 전송을 포함한 모든 실행 효과들은 되돌려집니다.

AWallet의 5번째 줄은 생성자이고, 컨트렉트가 생성될 때 한번 실행이 됩니다. Pay 함수는 wei 단위로 컨트렉트에서 받는 사람에게 양만큼 보냅니다. 컨트렉트의 8번째 줄에서 만약 caller가 owner가 아니거나 만약 소량의 ether가 실행에 사용되고 컨트렉트로 보내어졌다면, 그 컨트렉트는 예외를 throw하게 됩니다. 9번 째 줄에서, 만약 요구되는 ether의 양이 충족이 되지않는다면 그 call은 멈추게 됩니다. 10번 째 줄에서, 그 컨트렉트는 ether를 받는 곳에 보내기 전에 outflow 레지스트리를 업데이트합니다. 11번 째 줄에 보내는데 문제가 생긴 것들을 보여주는 목적으로 사용된 send 함수가 있습니다. 그것은 받는 곳이 contract일 경우에 실패할 것입니다.

**실행 비용들**. 각각 함수의 invocation은 이더리움 네트워크 안에서 모든 마이너들에 의해 이상적으로 사용되었습니다. 마이너들은 함수들을 invoke 하는 유저들에 의해 지불되는 실행 비용들에 의해 일을 하기 위한 인센티브를 받게됩니다. 게다가 인센티브들로 사용이 되고 있는, 실행비용들은 상대방이 시간-소비 계산들을 요구함으로써 네트워크 속도를 느리게 만드는 denial-of-service 공격들을 방어하는 수단이 됩니다.

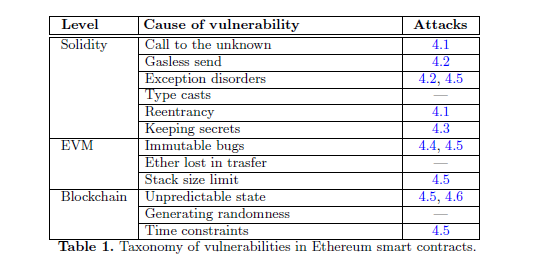
실행 비용들은 gas와 gas price의 용어들로 정의가 되고 그들의 product는 코드를 실행하기 위해 그 유저에 의해 지불되는 비용을 나타냅니다. 더욱 구체적으로, 호출을 발생시키는 트랜잭션이 유저가 기꺼이 지불을 하는 gas limit와 gas 단위 당 가격을 결정합니다. 각각의 gas 가격이 높아질수록, 마이너들이 그 트랜잭션을 실행하기 위해 선택하는 기회가 더 높아집니다. 각각의 EVM 작동은 gas의 특정 양을 소비하고 전체적인 비용은 마이너들에 의해 실행되는 작동들의 연속에 달려있습니다.

예외가 throw되지 않는 다면, 마이너들은 트랜잭션의 일반적인 정지할 때까지 트랜잭션을 실행합니다. 만약 트랜잭션이 성공적으로 정지하게 된다면, 남아있는 gas는 그 호출자에게 돌려줍니다. 그렇지 않다면 트랜잭션에 할당된 모든 gas는 잃습니다. 만약 computation이 할당된 모든 gas를 소비한다면, “out-of-gas” 예외- (호출자가 모든 gas를 잃는다.)로 정지하게 됩니다. Denial-of-service 공격을 하려는 상대방이 많은 양의 gas를 할당해야하고, 해당하는 ether를 지불해야합니다. 만약 그 상대방이 시장에 따라서 gas 가격을 선택한다면, 마이너들은 그 트랜잭션을 실행할 것이지만, 그 공격을 하기 위한 비용이 굉장히 많이 들게될 것입니다. 만약 그 가격이 너무 낮다면, 마이너들은 그 트랜잭션을 실행하지않을 것입니다.

**마이닝을 하는 과정.** 마이너들은 유저들에 의해 보내어진 트랜잭션들을 블록들에 그룹지어 넣게되고, 연관된 비용들을 모으기 위해 블록체인에 블록들을 추가하려고 시도합니다. Validity라고 부르는 조건들을 만족하는 블록들이 블록체인에 추가되게 됩니다. 이러한 조건들 중 하나는 일반적으로 이전 블록과 새로운 블록 안의 트랜잭션들에 달려있는 어려운 “작업증명” 문제를 푸는 것을 요구합니다. 그 문제의 difficulty는 1블록당 12초 정도의 마이닝 비율을 유지하기 위해 동적으로 업데이트됩니다.

한 마이너가 문제를 풀고 새로운 유효한 블록을 네트워크에 방송했을 때, 다른 마이너들은 그들의 시도들을 취소하고, 새로운 블록을 추가함으로써, 블록체인의 로컬 copy를 업데이트합니다. 그리고 블록체인의 가장 위에서 “마이닝”을 시작합니다. 문제를 푼 마이너는 새로운 블록 안에 있는 트랜잭션들의 실행비용들을 보상받게됩니다.

두명의 마이너들이 문제를 동시에 푸는 것이 아마 일어날겁니다. 이러한 경우에는, 블록체인은같은 부모 블록을 가리키는 새로운 블록을 가진 두 개의 가지들로 fork 하게됩니다. 합의 알고리즘은 마이너가 가장 긴 가지를 연장하도록 규정합니다. 비록 두 가지들이 계속해서 존재를 하게될 지라도, 결국에는 그 fork는 가장 긴 가지로 결정이 나게 됩니다. 그 트랜잭션들은 블록체인의 한 부분이 되지만, 가장 짧은 가지의 트랜잭션들은 제거가 될 것입니다. 그리고 그 보상 메커니즘은 full 비용들을 가장 긴 가지에서의 블록들의 마이너들에게 할당을 하고, 비용들 중 일부분은 제거된 가지의 root를 마이닝한 마이너들에게 할당합니다. A와 B 블록들이 같은 부모를 가지고 있고, 한 miner가 A의 상위에 새로운 블록을 추가했다고 가정해봅시다. 그 마이너는 A의 가지의 weight를 증가 시키기 위해 “uncle block”인 B의 miner에게 트랜잭션의 보상의 일부분을 기부할 수 있습니다.



**Solidity를 EVM 바이트코드로 컴파일하는 것.** 비록 컨트렉트들이 Solidity에서 함수들의 세트들로 표현이 될지라도, EVM 바이트코드는 함수들을 서포트하지 않습니다. 그러므로, Solidity 컴파일러는 그들의 첫 부분이 메커니즘을 실행하는 함수를 구현하기 위해 컨트렉트들을 해석합니다. 더욱 세부적으로, 각 함수는 함수의 이름과 타입 파라미터들을 기반으로 두고 있는 signature에 의해 식별이 됩니다. 함수 호출에 따라서, 이 signature은 호출된 컨트렉트 input으로 들어가게 됩니다. 만약 이 signature가 몇몇 함수와 매치가 된다면, 그 실행은 그에 부합하는 code로 점프를 하게 됩니다. 그렇지 않다면 signature은 fallback 함수로 점프하게 됩니다. 이것은 이름과 인수들이 없는 임의로 프로그래밍 된 함수입니다. Fallback 함수는 컨트렉트가 빈 signature로 전달될 때, 실행이 됩니다. 이것은 컨트렉트에 ether를 전송할 때 일어나게 됩니다.

Solidity는 ether를 전송하기를 허락하는 다른 컨트렉트에서 한 컨트렉트를 호출하기 위해 세 가지의 다른 construct들을 가집니다. 이러한 construct들은 같은 바이트 코드 명령어를 사용해서 컴파일 됩니다. 그 결과는 같은 행동도 여러 방식들로 구현이 될 수 있습니다. 미묘한 차이들은 Section 3에서 구체적으로 말하겠습니다.

**3. 스마트 컨트렉트들에서 취약점들의 분류**

이번 Section에서 우리는 이더리움 스마트 컨트렉트들의 보안 취약점들을 구조화 시킬 것입니다. 우리는 Solidity, EVM 바이트코드, blockchain 세 가지 클래스들로 취약점들을 분류합니다. 더 나아가 우리는 짧은 코드를 통해 Solidity level에서의 각 취약점들을 설명할 것 입니다. 그리고 모든 취약점들은 컨트렉트들에서 돈을 훔치는 공격들을 야기하도록 익스플로잇 될 수 있습니다. 테이블 1은 Section 4에서 설명할 공격들로 이어지는 취약점들의 분류를 정리 해놓은 것입니다.

**Call to the unknown.** Solidity에서 함수를 호출하고 ether를 전송하기 위해 사용되는 기본형들 중 몇 몇은 호출 된 곳/ 받는 곳의 fallback 함수 호출의 부작용을 가지고 있습니다. 우리는 아래에서 이러한 것들을 설명할 것 입니다.

-call은 함수를 호출하고 호출된 곳으로 ether를 전송합니다. E.g 아래처럼 컨트렉트 c의 함수 ping을 호출할 수 있습니다: **c.call.value(amount)(bytes4(sha3(“ping(uint256)”)),n);** 호출된 함수는 해쉬화 된 signature의 첫 4byte에 의해 식별이 됩니다. Amount는 wei가 얼마나 많이 c로 전송이 되었느냐를 결정하고, n은 실제 ping의 파라미터입니다. 만약 주어진 signature를 가지고 있는 한 함수가 주소 c에 존재하지 않는다면, c의 fallback 함수가 대신에 실행이 될 것입니다.

- send는 작동하고 있는 컨트렉트에서 받는 곳 r로 ether를 전송하기 위해 사용됩니다. Example) r.send(amount). Ether가 전송된 이후로 send는 받는 곳의 fallback 함수를 실행합니다. Send와 관련된 다른 취약점들은 “exception disorders”와 “gasless send”에 자세히 나와있습니다.

- delegatecall은 call과 유사합니다. 차이점은 호출 된 함수의 사용이 호출 한 곳의 환경에서 작동한다는 것입니다. 예를 들어서, **c.delegatecall(bytes4(sha3(“ping(uint256)”)),n)**을 실행한다고 해봅시다. 만약 ping이 this라는 변수를 받는다고 하면, 그 변수는 c의 주소가 아니라 호출자의 주소를 가리킵니다. 그리고 받는 곳 d로 ether가 전송이 되는 케이스에서 **– d.send(amount)를 통해 –** 그 ether는 호출자의 balance로부터 나오게 됩니다. ( section 4.6에서 공격을 보십시요.)

- 게다가 기본형들은 위와 같고, 한 번은 direct call를 아래 처럼 사용합니다. Contract Alice { function ping(uint) returns (uint) } Contract Bob { function pong(Alice c){ c.ping (42); } } 첫 번째줄은 Alice의 컨트렉트의 인터페이스를 선언한 것이고, 마지막 두번째 줄은 Bob의 컨트렉트를 포함하고 있습니다: pong은 direct call을 통해 Alice의 ping을 호출합니다. 만약 프로그래머가 Alice 컨트렉트의 인터페이스를 잘못 입력을 했다면( 파라미터 타입을 uint 대신에 int로 선언), Alice는 그 signature를 가지고 있는 함수를 가지고 있지 않습니다. 그러면 ping에 대한 호출은 결국 Alice의 fallback 함수를 호출 하게 됩니다.

Fallback 함수는 예측하지 못하게 실행되는 코드 부분만 있는 건 아닙니다. 9페이지에 있는 “type cast”와 11페이지에 있는 “unpredictable state”가 보고 된 다른 취약점들입니다.

**Exception disorder.** Soldity에서 예외가 발생되는 여러 상황들이 있습니다. e.g (i) 실행이 gas의 고갈 상태에서 동작될 때 (ii) call stack이 한계까지 도달했을 때 (iii) throw 명령이 실행될 때. 그러나, Solidity는 예외들을 다루는 방식에 있어서 단일화 되어있지 않습니다: 컨트렉트들이 서로를 호출하는 방법에 따라서 두 가지의 행동들이 있습니다. 예를 들어서,

Contract Alice { function ping(uint) returns (uint) } Contract Bob { uint x=0; function pong (Alice c){ x=1; c.ping(42); x=2; } }

지금, 몇몇 유저가 Bob의 pong을 호출하고 Alice의 ping이 예외를 throw한다고 가정해봅시다. 그러면 그 실행은 멈추고 그로 인해 일어난 전체 트랜잭션의 효과들이 되돌려집니다. 그러므로, 그 트랜잭션 이후로 필드 x는 0을 포함합니다. 지금, Bob가 ping을 call을 통해 호출한다고 가정합시다. 이러한 케이스에서, 그 호출로 인한 효과들만 되돌려집니다. 그 call은 false를 리턴하고, 실행은 계속됩니다. 그러므로, 트랜잭션 이후로 x는 2를 포함하고 있게됩니다.

더욱 일반적으로, 예외가 throw되었을 때, 중첩된 call들의 chain이 있다고 가정해봅시다. 그러면 예외는 다음과 같이 다뤄집니다.

-만약 체인의 모든 요소가 direct call이라면, 실행은 멈추고, 그로 인한 효과들은 되돌려집니다. 더나아가 모든 gas들은 소비된 트랜잭션을 되돌리는 것에 할당이 됩니다.

- 만약 최소한 체인의 한 요소가 call을 했다면, 예외는 호출된 컨트렉트들로 인한 모든 효과들을 되돌리면서 그 다음 call에 도달할 때까지 전파됩니다. 그러한 관점에서 실행은 유지되고, call은 false를 리턴 합니다. 더 나아가 모든 gas들은 소비 된 call에 의해 소진됩니다.

한 call에서 gas의 사용을 좀더 상위의 범주까지 사용하기 위해, 다음과 같이 씁니다.

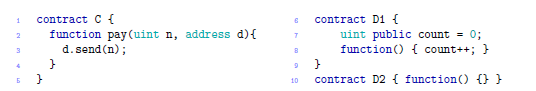
c.call.gas(g)(bytes4(sha3(“ping(uint256)”)),n);

만약, 범주가 구체화 되어있지 않는다면 모든 이용 가능한 gas는 잃게 됩니다. 그렇지 않다면 g만큼의 gas만 잃게 됩니다.

예외들이 어떻게 다뤄지는지 불규칙성은 컨트렉트들의 보안에 영향을 끼칩니다. 예를 들어서, ether의 전송이 성공적이라고 믿는 이유는 공격들로 이어지는 예외들이 없기 때문입니다. ( Section 4.2 와 4.5에 구체적으로 있습니다) 컨트렉트들의 28% 정도가 call / send 호출들의 리턴 값을 제어하지 않는 다는 것을 양적 분석을 통해 보여줍니다. ( 그러나 이러한 점검들의 부재가 필연적으로 취약점을 암시하는건 아닙니다. )

**Gasless send.** 한 컨트렉트로 ether를 보내는 send 함수를 사용할 때, gas 고갈 예외 상태에 빠지게 될 가능성이 있습니다. 이것은 프로그래머들에게 매우 예상치 못하게 다가왔습니다. 그 이유는 ether를 전송하는 것은 일반적으로 코드를 실행하는 것과는 연관이 없기 때문입니다. 이 예외 뒤에 감춰진 이유는 미묘합니다. 첫 번째로, c.send(amount)는 빈 signature를 가지고 있는 하나의 call의 같은 방식으로 컴파일이 되는 이유입니다. 하지만 피호출자에게 이용가는 한 gas의 수는 2300정도 입니다. 그러나, gas의 2300개는 그 컨트렉트의 상태를 바꿀 수 없는 제한된 바이트코드 명령들을 실행하도록 허락합니다. 다른 케이스에서, call는 결국 gas 고갈 예외 상태로 빠지게 됩니다.

우리는 send의 효과를 함수 pay와 두 곳의 받는 곳들인 D1,D2를 통해 ether를 전송하는 컨트렉트 C와 관련된 작은 예로 설명하겠습니다.



여기서 pay를 실행하기 위해 가능한 케이스들이 세 가지 있습니다.

-n ≠0 이고 d=D1일 때. C에서 send는 gas 고갈 예외에 실패합니다. 왜냐하면 2300개의 gas는 D1의 fallback 함수 상태를 업데이트하는 실행에 충분하지 않기 때문입니다.

- n ≠0 이고 d=D2일 때. C에서 send는 성공하게 됩니다. 왜냐하면 2300개의 gas는 D2의 빈 fallback 함수를 실행하는데 충분하기 때문입니다.

- n=0 이고 d ∈{ D1, D2 }일 때. 컴파일러 버전 < 0.4.0이면, C에서 send는 gas 고갈 예외에 실패하게 됩니다. 그 gas는 어떠한 fallback, 즉 심지어는 빈 fallback 함수도 실행하는데 충분하지 않기 때문입니다. 컴파일러 버전 >=0.4.0이면, 그 결과는 이전 두 케이스(d=D1 or d=D2)들 중 하나와 동일합니다.

요약하자면, send를 통해 ether를 전송하는 것은 두 개의 케이스들만 성공합니다.

1.받는 곳이 비싸지 않은 fallback 함수를 가지고 있는 컨트렉트. 2.받는 곳이 유저 일 때

**Type cases.** Solidity 컴파일러는 몇 개의 타입 에러들( 정수 값을 string 타입 변수에 할당하는 것)을 탐지합니다. 타입들은 direct call들에 사용이 됩니다: 호출자는 피호출자의 인터페이스를 선언을 해야만 하고, call이 실행되었을 때, 피호출자의 주소를 cast해야만 한다. 예를 들어서, ping에 대한 direct call을 다시 고려해봅시다.

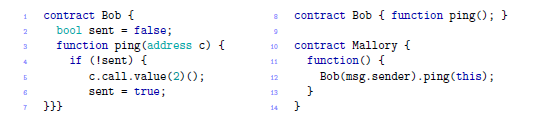
Contract Alice { function ping( uint ) returns (uint) } Contract Bob { function pong ( Alice c ) { c.ping (42); } }

Pong의 signature는 컴파일러에게 c가 Alice 인터페이스에 부착되었다는 것을 알려줍니다. 그러나 그 컴파일러는 그 인터페이스가 함수 ping을 선언했는지만 체크합니다. 반면에 아래와 같은 것들 것들은 체크하지 않습니다. (i) c는 Alice 컨트렉트의 주소라는 것 (ii) Bob에 의해 선언된 인터페이스가 Alice의 실제의 인터페이스와 매치하는 것. 외부의 type cast인 Alice(c).ping()에서도 일어납니다. C는 주소입니다.

한 컨트렉트가 타입 체크를 한다는 사실은 탐지된 (i)과 (ii)의 어떠한 에러든지 체크를 한다고 믿는 프로그래머들을 현혹하기 쉽습니다. 더 나아가, 심지어는 그러한 에러들의 존재에서, 그 컨트렉트는 실행 시간에 예외들을 throw하지 못합니다.그리고 direct call들은 call(예외들의 처리를 일 하기 위해 사용된 것과 같은 EVM 바이트 코드 명령으로 컴파일이 됩니다. 타입 미스매치의 경우에서 세 가지 다른 것들이 실행에 발생합니다.

-만약 c가 컨트렉트 주소가 아니라면, call은 어떠한 코드도 실행하는 것 없이 리턴합니다 -만약 c가 Alice의 ping과 같은 signature를 가지는 함수를 가지고있는 어떤 한 컨트렉트의 주소라고 한다면, 그 함수는 실행이 됩니다. – 만약 c가 Alice의 ping의 signature과 매칭이 되는 함수가 없는 컨트렉트라고 하면, c의 fallback 함수가 실행이 됩니다. 모든 케이스들에서 예외가 throw되는 경우가 없고 호출자가 에러를 인식하지 못합니다.

**Reenterancy.** 트랜잭션들의 원자성과 연속성은 비순환 함수가 호출되었을 때, 함수가 끝나기 전까지 재진입을 할 수 없다고 프로그래머들이 믿는 것을 유발합니다 그러나 이것은 항상 그렇게 되지않습니다. 왜냐하면 fallback 메커니즘이 공격자가 호출자 함수에 재진입을 가능하게 하기 때문입니다. 이것은 예상치 못한 작용들을 낳습니다. 그리고 모든 gas들을 소모하는 호출의 loop에 빠지게 만듭니다. 예를 들어서, 컨트렉트 Bob가 이미 블록체인에 있다고 가정해봅시다. 공격자가 Mallory 컨트렉트를 만들었을 때,



Bob에 있는 ping 함수는 빈 signature와 gas 제한이 없는 call을 사용해서 2wei를 주소 c에 보냈습니다. 이전에 언급된 것처럼 그 call은 ping을 다시 호출하는 Mallory의 fallback 함수를 호출 한것에 대한 부작용을 가지고 있습니다. 변수 sent는 이미 true로 세트되어 있지않기 때문에, Bob은 다시 Mallory에게 2wei를 보내고, 그녀의 fallback 함수를 다시 호출하면서, loop에 빠지게 된다. 이 loop는 그 실행이 결국 gas 고갈이 되었을 때, 또는 스택 한계에 도달했을 때, 혹은 Bob가 그의 ether가 모두 소모되었을 때 끝나게 됩니다. 모든 케이스들에서 예외는 throw됩니다. 그러나, call은 예외를 전파하지 않기 때문에, 모든 이전의 유효했던 ether의 전송들을 남겨둔채 마지막 call의 효과들만 되돌려집니다.

이 취약점은 ping 함수가 재진입이 되지않는다는 사실에 의해 야기되었습니다. 함수의 마침 전에 호출이 되는 것 같은 잘못된 작용을 합니다. 눈에 띄게도, 2016년 6월에 ether의 큰 손실을 야기 했던 “DAO 공격”은 이 취약점을 exploit한 것입니다.

**Keeping secrets.** 컨트렉트들에서 필드들은 모든 사람들에 의해 읽혀질 수 있는 public 혹은 다른 유저들/ 컨트렉트들에 의해 읽혀질 수 없는 private가 될 수 있습니다. 이것은 한 필드의 값을 설정하기 위해 유저들이 적절한 트랜잭션을 블록체인 위에서 공표하는 마이너들에게 보내야 합니다. 블록체인이 public하기 때문에, 모든 사람들이 트랜잭션의 내용물들을 검사할 수 있고, 그 필드의 새로운 값을 유추해낼 수 있습니다.

그렇게 구현하고 있는 멀티플레이어 게임들의 많은 컨트렉트들의 몇몇 필드들은 계속해서 비밀을 유지하는 것을 필요로 합니다: 예를 들어서, 한 필드가 한 플레이어의 다음 무브를 저장한다고 하면, 다른 플레이어들에게 무브를 알려준다는 것은 그들에게 그들의 다음 무브를 정하는 것에 있어서 이점을 줄 것입니다. 이러한 케이스들에서 한 필드가 특정 이벤트가 일어날 때까지는 비밀로 남겨두는 것을 보장하기위해서, 그 컨트렉트는 적절한 암호 기술을 exploit해야만 합니다. 마치 timed commitments같은 것처럼.

**Immutable bugs** 한 컨트렉트가 블록체인 위에 올려진다면, 그것은 더 이상 변경될 수가 없습니다. 그래서 유저들은 그 컨트렉트가 그들의 의도된 기능을 구현한다는 것을 신뢰할 수가 있습니다. 그리고 그것의 실행은 합의 프로토콜에 의해 보장되기 때문에, 예상대로 실행이 잘 될 것입니다. 결점은 만약 한 컨트렉트가 버그를 포함한다면, 그것을 곧바로 패치할 방법이 없다는 것입니다. 그래서 프로그래머들은 컨트렉트의 구현에서 한 컨트렉트를 바꾸거나 멈추기 위한 방법들을 모색해야합니다.-비록 그것이 이더리움의 원칙에 어긋나는 결합일지라도.

그 변경할 수 없는 버그들은 다양한 공격들로 exploit 될 수 있습니다. E.g ether를 훔치거나 다른 유저들에 의해 되돌릴 수 없도록 만드는 그런 공격들. ( Section 4.4 / 4.5를 보세요.). 이러한 모든 공격들에서, 회복의 가능성이 없습니다. 유일한 예외는 “DAO attack”에서의 회복입니다. 그 개응책은 공격과 관련된 트랜잭션들의 효과를 무효로 하는 블록체인의 hard-fork입니다. 이 해결책은 “ 코드는 법이다 “라고 명시된 원칙에 어긋나기 때문에, 이더리움 전체의 커뮤니티에 의해 동의된 것은 아닙니다. 결론적으로, 마이너들 중 일부분은 블록체인을 fork 하거나 대체 블록체인을 만드는 것을 반대하고 있습니다.

**Ether lost in transfer.**

이더를 전송할때 160비트 형식을 가진 주소를 명시해야합니다 이러한 주소들 중 많은 주소들이 고아입니다. 즉 어떠한 유저나 컨트렉트에도 관련이 없는 주소라는 것입니다. 만약 약간의 ether를 고아 주소로 보낸다면 그 이더는 영원히 잃게 됩니다. 잃은 이더는 되찾을 수 없기 때문에 프로그래머들이 받는 주소의 정확성을 보장해야됩니다.

**Stack size limit**

한 컨트렉트가 다른 컨트렉트를 호출 할때마다 트랜잭션과 연관된 스택 사이즈는 한 프레임씩 올라갑니다. 스택 사이즈는 최대 1024 프레임입니다. 스택 사이즈 한계일 때 더 많은 호출은 예외를 throw 합니다. 2016년 10월 18일까지 이대로 공격을 야기하는 사실을 가지고 exploit 하는 것이 가능했습니다. 상대방이 꽉찬 콜 스택으로 만들기 시작하고 그가 더 많은 호출에 실패할 희생자의 함수를 호출합니다. 만약 예왜가 적절하게 희생자의 컨트렉트에 의해 처리되지않는다면

그 상대방은 그의 스택을 조정할 수 있습니다. 이 취약점이 다른 취약점들과 함께 exploit 될 수 있습니다. Section 4.5에서 우리는 exception disorder와 스택사이즈리미트 취약점들을 익스플로잇 함으로써 악의적인 컨트렉트를 구현했습니다.

이 취약점의 원인은 이더리움 블록체인의 하드포크의 의해 생겨났습니다. fork가 EVM 명령들의 비용을 바꾸고 call과 delegatecall의 gas를 계산하는 방식을 재정의해버립니다.

fork 이후로 호출자는 호출 gas의 63/64 만큼을 할당합니다. 현재 블럭당 gas limit은 4.7M입니다. 이것은 최대 call stack가 1024보다 작다는 것을 말해줍니다.fork 이후로 호출자는 호출 gas의 63/64 만큼을 할당합니다. 현재 블럭당 gas limit은 4.7M입니다. 이것은 최대 call stack가 1024보다 작다는 것을 말해줍니다.

**Unpredictable state**

한 컨트렉트의 상태는 그것의 필드들과 balance의 가치에 의해 결정됩니다. 일반적으로 한 유저가 몇몇 컨트렉트를 호출하기 위해 트랜잭션을 네트워크에 보낼 때 그는 그 트랜잭션이 같은 상태에서 그 트랜잭션을 보낸 시간에 있는 컨트렉트를 실행한다는 보장을 할 수가 없습니다. 이것은 다른 트랜잭션들이 컨트렉트 상태를 바꾸기 때문에 일어납니다. 비록 그 유저가 한 트랜잭션을 처음으로 보낸다고 하더라도, 그런 트랜잭션이 처음으로 실행된다는 보장을 할 수 가없습니다.사실상 마이너들이 트랜잭션들을 블록들에 묶어 넣을 때 그들은 어떠한 요구도 지켜져야한다는 것이 요구되지않습니다. 그들은 몇몇 트랜잭션들을 포함하지 않게 선택을 하지 않을수도 있습니다. 한 유저가 그의 트랜잭션이 실행될 실제의 상태를 알지 못하는 또다른 상황이 있습니다. 이것은 블록체인 fork하는 상황에서 일어납니다. (Section 2를 보세요). 두 명의 마이너들이 새로운 유효 블록을 동시에 발견했을 때, 그 블록체인은 두 가지들로 fork 됩니다. 몇몇 다른 마이너들이 다른 가지들에 추가하는 동안, 몇몇 마이너들은 가지들 중 하나에 새로운 블록을 추가하려고 할 것입니다. 시간이 흐른 후에, 가장 긴 가지가 블록체인의 가지로 간주되는 반면에, 짧은 가지는 제거가 됩니다. 짧은 가지의 트랜잭션들은 무시됩니다. 왜냐하면 블록체인의 부분이 더 이상 아니기 때문입니다. 그러므로 , 한 컨트렉트가 특정 상태에 있다는 것을 믿는 것은 새로운 트랜잭션들을 만들기 위한 유저들에 대한 결정 요인입니다. 그러나 이후에, 그러한 상태는 되돌려집니다. 왜냐하면 블록체인에 이어져있던 트랜잭션들이 한 fork의 가장 짧은 가지에 있었기 때문입니다.

몇몇 케이스들에서 한 트랜잭션이 실행할 상태를 모른다는 것은 취약점들을 일으킬 여지를 주는 것입니다. 이것은 동적으로 업데이트 되는 컨트렉트들을 호출하는 케이스입니다. 비록 블록체인 위에 있는 한 컨트렉트의 코드는 변경될 수 없지만, 생각해보면 요소가 컨트렉트 주인의 요구에 따라 업데이트 될 수 있도록 한 컨트렉트를 옮기는 것은 가능합니다. 나중에, 그 주인은 호출자의 ether를 훔치는 악의적인 요소에 컨트렉트를 연결할 수 있습니다. ( Section 4.6의 공격을 보세요.)

**Generating randomness**

EVM 바이트코드의 실행은 결정적입니다: 잘못된 실행의 부재에서 한 트랜잭션을 실행하는 모든 마이너들은 같은 결과들을 가질 것입니다. 비-결정적인 선택들을 구현하기 위해, 많은 컨트렉트들은 초기화 된 seed가 모든 마이너들에게 각각 다르게 선택되는 pseudo-random 숫자들을 만듭니다.

이러한 seed를 가지기 위한 공통의 선택은 블록의 hash와 timestamp입니다. 모든 miner들이 그 블록체인의 같은 view를 가지기 때문에, 실행할 때 이 값은 모든 사람에게 같습니다. 명백하게, 이것은 미래 블록들의 내용이 예상될 수 없는 것처럼 random 숫자들을 만드는 안전한 방법입니다. 그러나 miner들이 어느 컨트렉트들을 블록에 넣을지 제어하기 때문에, 악의적인 miner가 pseudo-random의 결과를 통해 그의 블록을 옮기려고 시도할 수 있습니다. 비트코인 블록체인의 randomness에 대한 분석은 그 네트워크의 mining 파워의 대부분을 장악한 한 공격자가 50 비트코인을 결과의 분산 가능성을 통해 투자할 수 있다는 것을 보여줍니다.

이 문제에 대한 해결방안들은 시간이 걸리는 실행 프로토콜에 기반합니다. 이러한 프로토콜들에서 각각 참여자는 비밀을 선택하고, 그것의 요약을 보장에 대한 예금을 지불하면서 다른 사람들과 소통할 수 있습니다. 이후에 참가자들은 그들의 비밀들을 밝히거나 혹은 그들의 예금들을 잃는 것 중 선택해야합니다. 그 pseudo-random 숫자들은 모든 참가자들의 비밀들을 합침으로써 계산될 수 있습니다. 이러한 케이스에서 상대방은 결과를 통해 비밀을 밝히지 않을 수 있습니다. 그러나, 그렇게 하는 것은 자신의 예금을 잃는 결과가 있게 됩니다. 그 프로토콜은 비밀을 밝히지 않기 위해 예금의 양을 정할 수 있습니다.

**Time constraints**

어플리케이션들의 넓은 범주가 현재의 상태에서 action들이 허가되도록 결정하기 위한 time constraint들을 사용합니다. 전형적으로 time constraint들은 모든 miner들에 의해 합의된 블록 타임스템프들을 사용함으로써 구현됩니다.

컨트렉트들은 블록이 마이닝된 타임스탬프를 받을 수 있습니다. 한 블록 안에 있는 모든 트랜잭션들은 같은 타임스탭프를 공유합니다. 이것은 실행 이후의 컨트렉트 상태와의 결합으로 보장합니다. 그러나 그것은 새로운 블록을 만드는 miner가 임의의 특정 시간을 가진 timestamp를 선택할 수 있기 때문에 한 컨트렉트를 공격자들에게 노출시킬 수 있습니다. 만약 한 miner가 한 컨트렉트에 대한 지분을 가지고 있다면, 그는 그가 마이닝한 블록에 대한 적절한 타임스템프를 선택함으로써 이득을 얻을 수 있습니다. Section 4.5에서 우리는 이 취약점을 exploit한 공격을 보여줄 것입니다.