Alma Mater Studiorum · Università di Bologna

Scuola di Scienze
Corso di Laurea triennale in Informatica

Sicurezza, attacchi e best practice nei contratti Solidity di Ethereum

Relatore:
Chiar.mo Prof.
Cosimo Laneve

Presentata da:
Michele Emiliani

II Sessione
Anno Accademico 2017-2018

Indice

Capitolo 1

Introduzione sulla blockchain

1.1 Una tecnologia avveniristica

Viviamo in un epoca di grandi cambiamenti e rivoluzioni, alcuni a volte così repentini che ne abbiamo piena consapevolezza e comprensione quando sono già largamente diffusi. Nel mentre altri sono alle porte in un continuo ed esponenziale mutamento. Il mondo dell'informatica incarna esso stesso questo concetto più di ogni altra disciplina umana e da un anno all'altro software, hardware e tecniche diventano tanto obsolete come se appartenessero al secolo scorso. All'interno di questo grande dinamismo però, si possono evidenziare punti particolarmente importanti che, per la rilevanza che hanno non possono passare inosservati e che portano con sè la nascità, più che di una tecnologia, di una macro categoria che a volte si espande fino a diventare una disciplina a sè stante. In questa tesi trattiamo delle crypto currencies che, insieme alla tecnologia blockchain, si configurano come una delle più importante ed attuali rivoluzioni nel campo dell'informatica influenzando anche l'economia, la politica, i rapporti socio-lavorativi e in generale le stesse relazioni fra essere umani.

1.2 La nascita del Bitcoin

Già dagli anni '80 si è provato a creare un qualcosa che scindesse l'utilizzo, la gestione e il mantenimento di una moneta rappresentante un valore dagli antichi ordini che l'hanno sempre controllata, enti statuari molto centralizzati che ne decidono al pari di una proprietà privata o un titolo della propria azienda. È solo però dal 2008 tramite il famoso white paper sul Bitcoin scritto da Satoshi Nakamoto [1] (nome che ad oggi ancora non é stato associato ad una persona reale) che è stata definita la tecnologia blockchain come oggi la conosciamo. Questa è nata grazie alla ben progettata unione di elementi che già erano stati teorizzati e testati, grazie anche all'esperienza derivata dai precedenti progetti di monete virtuali, sopratutto dai loro errori e fallimenti.
Da lì in poi, in pochissimo tempo, è nata una branca a sè stante e ciò che poteva sembrare a prima vista una tecnologia utile solo per utopiche e poco pratiche crypto monete virtuali, si è rivelata essere le solide fondamenta sulle quali costruire un enorme numero di servizi, con una modularità quasi infinita e con proprietà fino a

quel momento ritenute impossibili o incredibilmente più dispendiose e difficili da implementare con metodi alternativi. Seguendo Bitcoin [2] sono nati quindi più di 1000 progetti in questi ultimi 10 anni che hanno visto coinvolte nazioni, grandi aziende ed enormi investimenti nell'ordine di miliardi di dollari effettuati da colossi dell'industria come Google, Amazon, Microsoft ma anche da banche e altre grandi multinazionali non legate strettamente al mondo dell' informatica.

1.3 Cenni sulla funzionamento della blockchain

Non analizzeremo in questa tesi nello specifico né il funzionamento dettagliato della blockchain e nemmeno gli algoritmi che risolvono il problema del consenso distribuito, notoriamente difficile in ambito informatico, o le differenze tra questi. La ragione è che esistono svariati paper, articoli e pagine web che fin dal 2008 trattano l'argomento ed illustrano come la blockchain sia: da un punto di vista teorico e pratico praticamente inattaccabile e che quindi ogni tipo di attacco diretto e basato sulla manomissione o modifica "criminale" di questo database distribuito sia futile e di fatto impraticabile.
Ciò di cui voglio trattare invece riguarda gli aspetti di sicurezza legati all'uso di alcune tecnologie fondate su una blockchain, le quali danno chiaramente già per scontato il suo funzionamento corretto, e più nello specifico l' esecuzione di smart contract, contratti automatici che definiscono pubblicamente e in maniera univoca una relazione di qualsiasi sorta tra due individui o enti.
Nonostante quindi non sia il nostro focus, ai fini della comprensione di questa tesi è necessario comunque descrivere in linea generale i meccanismi di funzionamento della blockchain. Il più grosso problema da risolvere negli anni dell' informatizzazione delle transazioni monetarie "classiche" è stato quello del
"double spending": cioè essere sicuri che non fosse possibile poter spendere due volte lo stesso quantitativo di denaro e questo è stato risolto attraverso rigidi controlli e costose, complesse e pesanti infrastrutture.
La blockchain, di default, non rende possibile questo scenario: essendo di fatto impossibile da manomettere, ogni utente, legato ad un address, è anche legato allo storico di tutte le sue transazioni, visibili pubblicamente, e si verifica, partendo dal primo blocco della stessa, che l'utente che provi a spendere la cifra X di denaro abbia come saldo, ottenuto aggiungendo le transazioni ricevute e sottraendo quelle inviate, almeno X.
L'impossibilità di manometterla invece si basa sul fatto che questa è un database distribuito tenuto online da molteplici nodi della quale ognuno di essi possiede una copia parziale o totale.
Un numero di nodi molto grande fa sì che per modificare ciò che è percepita come l' immagine della blockchain "vera" o "corretta" si debbano manomettere simultaneamente più del 50% di essi e visto il fatto che potrebbero trovarsi in diversi paesi e continenti ciò già è difficile. In aggiunta, la blockchain come dice il nome appunto, è una catena di blocchi, questi blocchi non sono che contenitori di transazioni, un nodo "attivo" nella rete è denominato "miner" e si occupa di aggiungere nuovi blocchi, contenenti nuove transazioni, in coda alla blockchain guadagnando-

ci in fee (mance), prima però di poter essere aggiunto per questo nuovo blocco si deve risolvere, per quanto riguarda la modalità di consenso attuale di Bitcoin e di Ethereum detta proof-of-work, un puzzle di hash *(il quale scala con difficoltà crescente al potere computazionale totale della rete)* e integrare nella soluzione l'hash del blocco precedente, calcolato a suo tempo alla stessa maniera.
Questo fa sì che per modificare un blocco nella blockchain si debbano per forza modificare anche tutti quelli aggiunti successivamente a questo ultimo e ciò, unitamente alla distribuzione su larga scala, rendono l'ipotesi di un attacco simultaneo che modifichi più del 50% delle copie online di essa fattivamente impraticabile.

I problemi di sicurezza però, nonostante questo fatto, ci sono e sono tanti, e per quanto non relativi al funzionamento stesso della blockchain, che è dimostrato funzionare senza vulnerabilità *(tranne alcuni scenari molto particolari che non andremo ad analizzare)*, nascono dalle strutture che ci vengono costruite sopra che più crescono in complessità più sono inclini ad avere possibili comportamenti anomali e non gestiti che potrebbero avere effetti indesiderati: il più temibile di tutti appunto rendere in qualche modo possibile una forma di double spending (e ciò si è verificato più di una volta).

Capitolo 2

Il network Ethereum

2.1 Storia e utilizzo

All'interno di questa analisi ci concentreremo solamente sul network Ethereum [3] [4], una piattaforma di sviluppo per applicativi decentralizzati e smart contract, la cui currency, cioè la moneta con cui vengono venduti e acquistati servizi all'interno di essa, è la seconda per volume di denaro tra tutte le crypto-valute, seconda solo a Bitcoin ed è molto probabile che, a lungo termine, la superi visto il ratio con cui si assottiglia la differenza tra BTC e ETH (nomi delle currency rispettivamente di Bitcoin ed Ethereum).
Ciò che rende questo progetto particolarmente interessante, al di là della sua notorietà e valore in denaro, è che non si tratta semplicemente di un servizio che fa un qualcosa di specifico, ma è più un gigantesco framework nonchè server per contratti e DAPP. Queste ultime sono Decentralized APPs simili in tutto e per tutto (nel lato client) ad APP tradizionali, come potrebbero essere siti o servizi web, ma con un diverso back-end appoggiato, invece che a database e server proprietari o noleggiati, sulla stessa blockchain ethereum. Queste godono dei benefici che ne derivano quali la sicurezza che i dati rimangano per sempre online e non possano mai essere cambiati, corrotti o eliminati, l'atomicità garantita delle operazioni, la sicurezza nelle transazioni data dal consenso e il fatto di non dover dipendere da nessuna autorità o ente/servizio privato. Rispetto a un alternativa classica i costi di mantenimento sono molto ridotti.

2.2 Gli smart contract

Questa comunicazione tra le DAPP e la blockchain è definita da smart contract anch'essi salvati sulla blockchain. Un contratto è una funzione automatica sempre in esecuzione, identificata da un indirizzo tramite il quale è possibile chiamarlo, da un codice, salvato in bytecode sulla blockchain che ne definisce il comportamento, e da un saldo che contiene i proventi delle transazioni inviate all'indirizzo associato al contratto e dal quale vengono detratti gli ETH necessari a pagare il gas che è stato utilizzato nell'esecuzione delle sue funzioni.
I contratti sono visibili in chiaro e pubblicamente sulla blockchain e rendono trasparente, sempre funzionante e impossibile da interrompere, sia in buona che in

cattiva sorte, un comportamento o una reazione/risposta dato il verificarsi di un evento o di determinate circostanze.

E' evidente il vantaggio pratico dell'utilizzo di entrambi: dal punto di vista dei contratti un datore di lavoro potrebbe scrivere il contratto a un suo dipendente sulla blockchain Ethereum e il contratto stesso alla fine del mese si occuperebbe in automatico, al costo di una fee in gas solitamente molto bassa, di togliere la cifra pattuita dal saldo del contratto, eventualmente chiamandone un altro che faccia un versamento sullo stesso, e darla al dipendente tramite una transazione, senza dover passare per banche, tasse per bonifici, o anche l'eventuale malafede del datore di non voler pagare il proprio dipendente. Vista però l'impossibilità di bloccare il contratto, anche nel caso in cui entrambi le parti lo volessero, è necessario inserire una funzione di uscita o di distruzione (selfdestruct() funzione standard Solidity), su accordo delle parti, come metodo per fermarlo. Se questa non fosse inserita manualmente nel contratto, questo continuerebbe la sua esecuzione all'infinito o fino al termine degli ETH nel saldo del contratto, ipotizzando di non avere più transazioni a favore di questo.

2.3 Le DAPP

Le DAPP invece hanno i seguenti vantaggi: invece di dover creare la propria blockchain e le proprie currency autonome si appoggiano sulla blockchain Ethereum già esistente e consolidata così da risparmiare risorse che sarebbero necessarie nel crearne una propria, ci guadagnano in sicurezza poiché la rete è già testata, avviata e possiede già un grande numero di nodi e nell avere un token, che risulta essere una moneta a sè stante, ma che possiede lo stesso standard (ERC20) di tutti gli altri token
Ethereum e ciò ne facilita l'utilizzo sia nell' implementazione che nella gestione poiché lo rende istantaneamente compatibile con tutti i contratti e gli applicativi wallet che supportano lo standard ERC20 di Ethereum e le transazioni tra token di questo tipo. La differenza tra un token e una currency è che il primo non possiede una blockchain ad esso dedicata ma si appoggia su un'altra, nel caso dei token con standard ERC20, quella di Ethereum, la cui currency ha anch'essa questo standard.

2.4 I client

Il browser Mist, ancora in fase di forte sviluppo, si occupa di fare da marketplace e showcase delle DAPP, scaricandolo si possono usare i servizi ethereum e ci si sincronizza con la blockchain, diventando a tutti gli effetti un nodo (passivo) della rete che ne possiede un backup, parziale o totale a propria scelta. Esistono anche altre implementazioni del client oltre quella ufficiale,
da programmi da riga di comando come:
go-ethereum,
cpp-ethereum,
pyethapp,

ethereumjs-lib,
ethereumJ,
ruby-ethereum,
node-ethereum,
Nethereum e
ethereum-haskell,
fino a servizi web o applicativi desktop ed estensioni per i browser come Parity e Metamask.
Esiste anche una libreria JavaScript "web3.js" che contiene un oggetto "web3.eth" con funzioni relative ad Ethereum e che comunica con un nodo locale attraverso chiamate JSON RPC, le sue funzioni di default usano richieste HTTP sincrone. Questa può essere particolarmente utile nel caso si stia sviluppando una DAPP.
Alcune di queste implementazioni permettono una sincronizzazione più veloce o la possibilità di accedere a un nodo ethereum già collegato alla rete quindi senza dover aspettare o scaricare nulla sul proprio terminale e potendo operare attraverso un nodo attivo e già perfettamente sincronizzato.

Capitolo 3

I linguaggi e il gas

3.1 Altre piattaforme per smart contract

Dopo aver introdotto in generale la rete Ethereum parleremo ora più nel dettaglio degli smart contract poiché le DAPP, pur appoggiandosi alla piattaforma, sono esse stesse servizi a parte con differenti obiettivi, dinamiche ed eventualmente linguaggi di programmazione usati e costituiscono di fatto un diverso ecosistema. Non ci soffermeremo sul confronto ma esistono tante piattaforme [5] che danno la possibilità agli utenti di creare smart contract:

- Bithalo, Spark e Rootstock ad esempio sono progetti relativi a Bitcoin
- CounterParty un porting della EVM sulla blockchain del Bitcoin, usa proof-of-burn e gli stessi linguaggi di Ethereum, i contratti sono scritti nel campo data di transazioni di bitcoin
- Lisk permette di scrivere contratti in JavaScript o node.js con un consenso dpos, ogni contratto ha una sua side-chain e può scegliere quali nodi partecipano per raggiungere il consenso
- Tezos punterà su un controllo formale automatizzato e su Michaelson un linguaggio ad-hoc
- EOS su una migliore scalabilità per il futuro tramite un esecuzione concorrente e molto veloce,
- Stratis permette di scrivere contratti in C#
- Antshares in C# ma anche in VB, .Net e F# e in futuro supporterà JAVA, C, C++, GO, Python e Javascript,
- Minereum propone contratti che si auto minano
- Monax supporta i contratti Ethereum e la creazione di blockchain private con relative regole per l'accesso
- Stellar implementa contract semplici ma con la particolarità di avere chain di transazioni e account multisignature cioè accessibili da più persone che richiedono un certo consenso tra i partecipanti per effetturare operazioni

e altre come Blackcoin, Namecoin, Mastercoin, Codius, DAML, Dogeparty, Symbiont e in passato anche Ripple che ora ha interrotto lo sviluppo.
Ethereum tra queste, nonostante non sia perfetta, è sicuramente quella più usata e rilevante nonchè quella su cui si è iniziato a costruire maggiormente.
Se nel futuro la situazione rimarrà invariata o una piattaforma, già esistente oppure no, prenderà piede e surclasserà Ethereum non lo possiamo sapere, tutto ciò che sappiamo è che questa tecnologia essendo ancora molto embrionale si evolve rapidamente e risulta perciò molto difficile se non impossibile fare delle previsioni al riguardo nel medio-lungo termine.

3.2 I linguaggi

Per scrivere uno smart contract esistono su internet varie linee guida, che però sono ancora troppo incomplete rispetto alle loro controparti più tradizionali, come quelle per scrivere programmi in linguaggi noti come C, C++, Python, Java, o per costruire siti web usando Html, Css, Javascript o rispettivi preprocessori/-framework. Ciò genera dubbi riguardo alle best practice da usare per evitare di incorrere in bug o comportamenti inaspettati e non gestiti, all' interno oltretutto di un settore molto critico in cui una singola cifra sbagliata potrebbe comportare perdite di milioni di dollari.

Esistono vari linguaggi di programmazione per scrivere smart contract Ethereum e in futuro questa lista crescerà inglobando anche, se sarà possibile, linguaggi molto noti e usati come Java o C. Quelli che ci sono ora sono stati creati ad-hoc per il sistema in quanto è risultato più veloce e meno complessa la loro implementazione rispetto all'adattamente di un linguaggio già preesistente. Si è scelto di creare linguaggi da zero che hanno in sè proprietà particolari in quanto sarebbe stata una sfida considerevole adattare linguaggi già esistenti, complessi e con le loro virtual machine dedicate, tante librerie e funzioni a questi nuovi meccanismi di programmazione, primo fra tutti per complessità il gas e la sua gestione.
In futuro comunque, con i prossimi roadmap è in programma di adattare la programmazione di contratti anche a questi linguaggi già noti e consolidati.
Questi linguaggi per smart contract vengono poi compilati e quindi tradotti in istruzioni bytecode per la Ethereum Virtual Machine e citandone alcuni troviamo:

- VIPER, un linguaggio simile a python fortemente tipato decidibile
- SERPENT, un linguaggio di basso livello sviluppato agli albori di ethereum come primo candidato
- LLL, un linguaggio ancor più di basso livello simile a Lisp
- MUTAN un linguaggio ispirato a Go ma deprecato e di cui è sconsigliato l'utilizzo
- Solidity

Tra questi quello che andremo ad analizzare è quest ultimo.
Solidity [30] è un linguaggio a oggetti compilato fortemente tipato ispirato a Javascript sviluppato ad-hoc dal team di Ethereum, presente su github ed è quello più

in vista, maggiormente sviluppato ed è, inoltre, l'unico di fatto ad essere supportato "ufficialmente" rispetto agli altri linguaggi che via via sono stati sempre più accantonati in favore di quest'ultimo.

3.3 La EVM e il gas

```
1  step          1     Default amount of gas to pay for an execution cycle
2  stop          0     Nothing paid for the SUICIDE operation
3  sha3          20    Paid for a SHA3 operation
4  sload         20    Paid for a SLOAD operation
5  sstore        100   Paid for a normal SSTORE operation
6  balance       20    Paid for a BALANCE operation
7  create        100   Paid for a CREATE operation
8  call          20    Paid for a CALL operation
9  memory        1     Paid for every additional word when expanding memory
10 txdata        5     Paid for every byte of data or code for a transaction
11 transaction   500   Paid for every transaction
```

Un paio di istruzioni EVM con il relativo costo in gas e descrizione, una send Solidity di default costa 2300 gas e più avanti approfondiremo cosa questo comporti.

Una volta scritto un contratto in Solidity, questo viene compilato in bytecode eseguibile dalla Ethereum Virtual Machine (EVM). La relazione con quest'ultima crea complessità in quanto chi sviluppa in Solidity deve anche avere ben presente il funzionamento della EVM poiché ogni istruzione che scrive non è che un superset di istruzioni (opcode) di più basso livello. Programmare in Solidity senza conoscere come una funzione venga tradotta e che proprietà abbiano le corrispondenti operazioni in bytecode è stato causa di numerosi bug e ad oggi è il primo e più importante aspetto da considerare.

L'elemento più pregnante in questo senso è il gas. Al fine di evitare attacchi DoS/DDoS la rete ethereum ha concepito questo espediente: nell'eseguire contratti ogni transazione o funzione costa un certo ammontare di gas che si può acquistare con degli ETH. In base a quante risorse (calcoli, memoria occupata, numero di accessi in memoria, etc) un operazione "consuma" per essere eseguita, aumenterà in proporzione il suo costo in gas.

Alcune funzioni Solidity hanno un certo costo in gas fissato derivante dal costo in gas sommato delle operazioni bytecode che le compongono, a volte però, all'interno di contratti complessi, per motivi inaspettati o poco prevedibili (esempio: la chiamata a un altro contratto il cui costo in gas è cambiato, non è costante o è stato calcolato male) il costo del'esecuzione del contratto può aumentare e nel caso in cui il gas richiesto superi la quantità di gas assegnato, deciso arbitrariamente dal chiamante, questo non completerà l'esecuzione. In questi casi viene fatto un rollback di tutte le operazioni eseguite dal contratto e ogni risorsa e cambiamento vengono riportati allo stato iniziale precedente l'esecuzione del contratto. Il gas consumato però viene perso in quanto il contratto, nonostante sia "fallito", ha comunque consumato tempo e risorse della rete. Nel caso invece ne fosse assegnato

di più del necessario, la differenza sarebbe ritornata al chiamante alla fine dell'esecuzione. Conoscere con buona approssimazione quanto gas possa consumare un contratto si è rivelato un compito non facile, ma molto utile per una numerosa serie di fattori. Uno fra tutti, banalmente: avendo risorse limitate e volendo eseguire più di un contratto, si vuole evitare sia di spendere troppo poco, e perdere gas in un tentativo fallito di esecuzione, sia di dover richiamare il contratto una seconda volta, oppure di assegnarne troppo ad un contratto solo e rimanere quindi con troppo poco gas per poterne chiamarne un altro.

3.4 Considerazioni economiche sul gas

Per concludere: è anche necessario puntualizzare che il gas, per quanto possa sembrare un token o una currency da questo discorso, di fatto invece non lo è, non è possibile acquistarlo, possederlo e non ha un prezzo o una valutazione di mercato (cosa che invece ETH, BTC e altre currency e token hanno). Questo perché il costo della computazione è bene che rimanga coerente con il costo effettivo della stessa e non avrebbe senso che questo cambiasse relativamente alle oscillazioni di mercato della currency ETH. Difatti per far rimanere l'intero sistema scalabile, mentre ETH può guadagnare o perdere valore come una qualsiasi moneta o titolo, il gas invece mantiene un valore del tutto scollegato da questa, tant'è che il suo prezzo è puramente arbitrario e deciso da ogni chiamante di un contratto e da ogni miner in forma di soglia minima.
Per puntualizzare brevemente, un miner è un nodo attivo nella rete che contribuisce in prima persona al suo sostentamento eseguendo sulla propria macchina le operazioni richieste per una chiamata ed esecuzione di un contratto o di una transazione, collezionando le fee associate a questi, calcolate in forma di gas.
Nel caso in cui un miner accetti di eseguire un contratto, significa che il chiamante di questo ha scelto di pagare un certo prezzo gas-ETH (detto gasPrice) e che questo prezzo è uguale o superiore al prezzo minimo a cui il miner ha deciso di "vendere" il proprio potere computazionale e in generale le proprie risorse.
Questo meccanismo permette che la computazione abbia sempre un prezzo sensato in base al suo costo effettivo in quanto un contratto con un gasPrice troppo basso non vorrà essere eseguito da nessuno, perché il guadagno sarebbe nullo o negativo rispetto ai costi dell'elettricità e del tempo, mentre uno con un gasPrice elevato sarà eseguito molto rapidamente in quanto tanti miner saranno in gara tra di loro per aggiudicarselo.
In questo caso però, solo uno vincerà la corsa e lo eseguirà, aggiungendolo nel blocco da lui creato e riscuotendone la fee associate (unità di gas usato * gasPrice) una volta che il blocco sarà aggiunto permanentemente alla blockchain. Inoltre se il valore del token ETH dovesse decuplicare, decuplicherebbe anche, di fatto, la quantità media di gas acquistata con un unità di ETH, mantenendo così in qualche modo sempre costante, o sensato rispetto al suo effettivo costo nella realtà, il prezzo della computazione.

Capitolo 4

Solidity: Bug e best practice

Analizzeremo ora nello specifico i bug più famosi di Solidity [6], quelli che hanno avuto più rilevanza storica e informatica, che hanno fatto aprire gli occhi rispetto ad alcuni rischi fino a quel momento non ritenuti meritevoli dell'attenzione dovuta e che tutt'ora rappresentano una possibile minaccia per qualsiasi creatore o utilizzatore di contratti. Essendo la blockchain immutabile, prima di parlare di ciò, è adeguato aggiungere che, pur non essendo questo un vero e proprio bug ma più una feature, una transazione è irreversibile e quindi nel caso si sbagli l'indirizzo di invio non esiste una procedura di recupero e gli ETH inviati sono persi per sempre o "morti" in quanto mai più utilizzabili da nessun altro.

Parliamo per prima dell'assenza di reentrancy, visto che poi ci servirà anche successivamente per le spiegazioni di altri bug.

4.1 Reentrancy in THE DAO

4.1.1 Storia

Il più famoso fra tutti i bug è sicuramente il bug di "THE DAO". Questo era un contratto con decine di migliaia di partecipanti che si occupava di raccogliere fondi in base a quanto ogni partecipante arbitrariamente decidesse di metterci e fungeva da fondo di investimento cioè, tramite votazioni regolamentate da un meccanismo di consenso simile a quello della blockchain, questi soldi venivano investiti in una qualche maniera al fine di produrre
guadagno di cui, in base alla percentuale messa, ogni investitore avrebbe beneficiato. Chiaramente in qualsiasi momento era possibile ritirare i soldi depositati tramite chiamate a certe funzioni del contratto stesso.
Fino a qui sembra che non ci sia nulla che non vada, il contratto era apparentemente scritto bene e la rete ethereum sicura per definizione, viste le proprietà della blockchain e della gestione del consenso. Il bug, infatti, era localizzato nel contratto e dipendente dal fatto che alcune funzioni non erano gestite a dovere ma questo era, per la maggior parte, reso possibile da un difetto intrinseco al linguaggio Solidity: una tipologia di struttura di codice che il compilatore non riconosce come errata e vulnerabile e che quindi non segnala, nonostante questa possa dare luogo ad esiti imprevisti e sicuramente non calcolati all'atto della stesura e revisione del contratto.

4.1.2 Spiegazione dell' exploit

Esamineremo adesso la funzione forse più critica di tutto il linguaggio, la SEND o altre implementazioni di questa, ovvero la funzione che si occupa di effettuare una transazione, cioè di trasferire ETH da un indirizzo all'altro. All' interno di questo contratto però, il fondo di un utente non era un collegamento al suo portafoglio personale ethereum (creato con myetherwallet, la DAPP wallet di Mist, o altre implementazioni), ma era l'array standard "balance" a contenere, in ogni rispettiva casella "balance[utente]", nota del saldo totale di deposito di quest'ultimo.

La funzione contenente la vulnerabilità era quella che serve a fare lo split, per maggiore chiarezza occorre spiegare più in dettaglio il funzionamento intrinseco di questo contratto.
Per poter prelevare fondi dal contratto, un utente deve prima proporre uno split, cioè proporre di creare un fondo figlio in cui, una volta accettata la proposta e creato lo stesso, spostare poi ciò che vuole ritirare. Una volta approvato lo split, il childDAO può essere estinto insieme ai soldi che contiene, cioè l'utente può "distruggere" il fondo figlio facendo ritornare i soldi che contiene ai legittimi proprietari. Nel caso questo contenga solamente soldi suoi, semplicemente ed effettivamente ritirandoli dal fondo.
Già all'epoca dell'attacco, il pattern errato che dava al contratto di The DAO e a parecchi altri smart contract ethereum questa vulnerabilità era già stato indentificato ed era ben conosciuto dalla parte più informata della community oltre che dagli sviluppatori di The DAO, tanto che il "disastro" che che causo la perdità di 60 milioni di dollari avvenne poche ore prima del già annunciato rilascio di una

patch che avrebbe modificato la sezione di codice contenente la vulnerabilità.
Il problema in questo caso è generato dalla "Reentrancy": una funzione è "reentrant" se può essere interrotta e ripresa in un secondo momento e, a prescindere da cosa può essere accaduto nel mezzo, il suo risultato finale / l'output che produce rimangono sempre e comunque deterministicamente i medesimi.
Le funzioni Solidity non posseggono già di partenza, indipendentemente da come vengono scritte, questa proprietà, e ciò lascia a chi scrive il codice il compito di farlo in maniera tale che le funzioni siano "reentrant", cosa che in questo caso non avvenne (in tempo).
Qui l'errore fu che la funzione, come si può vedere dal codice, era stata scritta in modo tale che PRIMA mandasse l'ammontare che si voleva ritirare, POI aggiornasse il saldo del conto sottraendo i soldi ritirati, ed era perciò possibile intromettersi a metà tra il ritiro e l'aggiornamento alterando così l'esecuzione [7] .

4.1.3 Il codice del contratto

La funzione "splitDAO" che conteneva la vulnerabilità per via dell'ordine delle operazioni:

```
1  function splitDAO( uint _proposalID, address _newCurator)
2  noEther onlyTokenholders returns (bool _success) {
3          /.../
4          // Muove ETH e assegna nuovi token.
5          //Notare che questa operazione viene fatta per prima
6          uint fundsToBeMoved =
7                  (balances[msg.sender]*p.splitData[0].splitBalance) /
8                  p.splitData[0].totalSupply;
9          if (p.splitData[0].newDAO.createTokenProxy
10             .value(fundsToBeMoved)(msg.sender)==false)
11         // La riga precedente è quella che l'attacker
12         //vuole eseguire più di una volta
13                 throw;
14         /.../
15         // Distrugge i token del DAO
16         Transfer(msg.sender, 0, balances[msg.sender]);
17         withdrawRewardFor(msg.sender); //La funzione che contiene la vulnerabilità
18         // Notare che la riga precendente è erroneamente
19         //posta prima delle successive due
20         totalSupply -= balances[msg.sender]; // Questo viene fatto alla fine
21         balances[msg.sender] = 0; // E anche questo viene fatto alla fine
22         paidOut[msg.sender] = 0;
23         return true;
24 }
```

Nella pagina seguente (20) la funzione "withdrawRewardFor", che contiene la chiamata alla funzione "payOut" con la chiamata vulnerabile a "_recipient" in quanto indirizzo non verificato:

```
1  // La funzione "withdrawrewardfor" nel dettaglio
2  function withdrawRewardFor(address _account)
3  noEther internal returns(bool _success) {
4          if ((balanceOf(_account)*rewardAccount.accumulatedInput()) /
5                  totalSupply < paidOut[_account])
6                  throw;
7          uint reward = (balanceOf(_account)*rewardAccount.accumulatedInput()) /
8          totalSupply-paidOut[_account];
9          // Ecco la vulnerabilità la funzione payout
10         if (!rewardAccount.payOut(_account, reward))
11                 throw;
12         paidOut[_account] += reward;
13         return true;
14 }
```

La funzione "payOut" nel dettaglio:

```
1  function payOut(address _recipient, uint _amount) returns(bool) {
2          if (msg.sender!=owner || msg.value>0 || (payOwnerOnly&&_recipient!=owner))
3            throw;
4          // La seguente è la riga vulnerabile per via della chiamata a "_recipient"
5          if (_recipient.call.value(_amount)()) {
6            PayOut(_recipient, _amount);
7            return true;
8          }
9          else
10                 return false;
11 }
```

Ritornando false tutte le condition presenti nel primo if, in quanto il contratto pensa di essere ancora allo stato iniziale, non viene eseguito il primo throw e quindi il secondo if viene sempre eseguito, questo contiene nella sua condition la chiamata vulnerabile *(riga 5 della funzione "payOut")* a "_recipient": "_recipient.call.value(_amount)()".
Quindi, durante l'esecuzione (ancora) della prima chiamata fatta a splitDAO, viene qui chiamato "_recipient" e cioè proprio il contratto dell' attaccante, e la chiamata in questione *(sempre riga 5 di "payOut")* come risultato fa si che, venendo chiamato, sia anche eseguito un contratto. Questo, volontariamente creato dall' attaccante per avere quest' effetto, semplicemente non fa che richiamare una seconda volta la funzione splitDAO, quella già chiamata in precedenza, utilizzando gli stessi parametri della prima chiamata. Essendo di fatto ancora all'interno dell'esecuzione di questa prima chiamata, l'ultima parte della funzione splitDAO, quella relativa alla prima chiamata di questa, rimane in "attesa" visto che il controllo viene passato a una seconda istanza di se stessa, invocata tramite il contratto dell' attaccante. In questa parte finale ancora da eseguire, però, si trovano le operazioni tramite le quali il saldo del fondo viene aggiornato.

4.1.4 Riassunto dell' attacco

L'attaccante in questo modo ha creato un loop di chiamate innestate, e continuando a interrompere l'aggiornamento del saldo di ogni chiamata con una successiva chiamata identica, dopo il ritiro dei fondi e prima dell'aggiornamento del

saldo, ha prodotto un loop in cui tutti gli aggiornamenti del suo saldo sono stati eseguiti alla fine di tutte le chiamate da lui fatte a splitDAO, in ordine inverso a quello di chiamata. Cioè, il suo saldo è stato posto a 0 dall'esecuzione dell'ultima chiamata (unica a eseguire senza essere interrotta), e poi rimesso a 0 n-1 volte via via in ordine inverso dalle precedenti chiamate che completavano la loro esecuzione. Al contempo però, per ognuna di queste chiamate innestate ha ritirato il suo saldo n volte [8] [9] [10].
In realtà poi, questo attacco avrebbe dovuto fruttare una cifra minore di quella ritirata, considerato il numero di chiamate fatte e i soldi presenti inizialmente nel saldo dell'attaccante. In teoria avrebbe dovuto ritirare solo una piccola parte del fondo (all'epoca di 150milioni, pari a circa il 20% di tutti i token ETH in circolazione), mentre ne ha ritirati quasi la metà, 60 milioni di dollari. Di fatto, invece, accadeva che a ogni chiamata, utilizzando a suo vantaggio la funzione "transfer", (una variante della classica send standard di Solidity, creata ad-hoc per il contratto The DAO) è riuscito a spostare i soldi che il suo childDAO possedeva nuovamente in esso effettivamente moltiplicando per 2, a ogni passaggio, i soldi che il suo child-DAO risultava possedere, e che quindi venivano mossi ad esso dal DAO principale al momento di estinguerlo e cioè di ritirare.
In questa maniera, ogni chiamata era in grado di spostare verso il child il doppio rispetto alla precedente (saldo)+(saldo*2)+(saldo*2*2)+(saldo*2*2*2) e così via n volte quanto il numero di chiamate innestate. L'incremento è stato così elevato che da relativamente poche centinaia di dollari, in un arco ristretto di tempo e utilizzando un solo computer, l'attaccante è riuscito a trasferire 60milioni di dollari all'interno del childDAO da lui creato per l' attacco.

Riassumendo:

1. Proponi uno split e aspetta finchè non finisce il periodo per votare. (DAO.sol, createProposal)

2. Esegui lo split. (DAO.sol, splitDAO)

3. Fai si che il DAO mandi al tuo nuovo DAO la sua parte di token. (splitDAO ->TokenCreation.sol, createTokenProxy)

4. Fai in modo che il DAO provi a mandarti il premio prima che aggiorni il tuo saldo, ma dopo aver eseguito il punto (3). (splitDAO ->withdrawRewardFor ->ManagedAccount.sol, payOut)

5. Mentre il DAO è al punto (4), fagli eseguire splitDAO di nuovo con gli stessi paramentri che al punto (2)
   (payOut ->_recipient.call.value ->_recipient())

6. Il DAO ti manderà più child token, ritira il tuo premio nuovamente prima che aggiorni il tuo saldo. (DAO.sol, splitDAO)

7. Riesegui il punto (5)!

8. Alla fine il DAO aggiorna il tuo saldo. Visto che il punto (7) ritorna al punto (5), questo non succederà mai fintanto che continui a fare chiamate.

Alla fine l'unica soluzione, fallita la proposta di un meno invasivo soft fork, è stata quella di un hard fork, cioè un cambiamento strutturale delle regole e dei meccanismi della stessa piattaform Ethereum, attraverso il quale l'indirizzo dell'attaccante, quello che possedeva i 60milioni, è stato congelato in modo da non permettergli mai di ritirare quegli ETH, annullando così di fatto il suo split e privandolo della proprietà di quei token, fattivamente cancellando le suddette transazioni dalla blockchain. In base poi alle azioni dell'attaccante e analizzando le sue scelte sia durante che dopo l'attacco, si capisce che il furto è stato fatto non tanto al fine di tenere il denaro per sè, cosa che avrebbe potuto fare facilmente agendo meno allo scoperto e non dichiarando pubblicamente il fatto, ma più per sottolineare i gravi rischi a cui un sistema come questo può portare e inoltre, non si sa se volontariamente o meno, ha evidenziato alcune contraddizioni anche nell'ideologia stessa di decentralizzazione quando, per risolvere la questione, si è dovuti ricorrere a un azione molto autoritaria e centralizzata, decisa e messa in atto dal creatore della piattaforma Vitalik Buterin in persona, oltre che mettere in dubbio principi fondamentali come l'immutabilità di un contratto.

Una soluzione molto dura e valida solo per questo caso specifico, la cui ripercussione è stata quella di dividere Ethereum in due: una parte che non ha accettato questo provvedimento e la modifica di un contratto, che per i principi stessi di Ethereum dovrebbe essere immutabile, ha fondato Ethereum Classic e
mantiene le regole precedenti; i restanti invece sono rimasti in Ethereum accettandole. Ciò ha creato, nella data del provvedimento, una scissione permanente e la nascita di una nuova currency ETC e relativa blockchain per Ethereum Classic, la quale è del tutto indipendente da quella di Ethereum, ma condivide gli stessi blocchi di questa per le date precedenti quella dell'hard fork. Ad oggi Ethereum contiene quasi 30 volte il capitale di Ethereum Classic.

Nonostante questo provvedimento, ogni contratto Ethereum e lo stesso "The DAO" sono ancora vulnerabili a questo tipo di attacco e l'unico modo per prevenirlo è, in contratti futuri, avere ben presenti queste dinamiche e tenere a mente fattori come la reentrancy, l'aggiornamento di un saldo prima di una send e non dopo, la chiamata a un contratto non definito, e in generale usare un costrutto che renda più atomica possibile la sezione contenente send e aggiornamento del saldo, magari anche tramite mutex visto che, nonostante i contratti e la rete ethereum non prevedano concorrenza, di fatto questa è un po' come se ci fosse, poiché è possibile effettuare chiamate a un qualsiasi contratto (sia di nuovo lo stesso o un altro anche arbitrario) anche durante l'esecuzione di uno stesso e ciò porta possibilmente a *race-conditions* e a problemi simili a quelli affrontati nell'ambito di programmi che vengono eseguiti in parallelo con o senza risorse condivise.

4.1.5 Esempi di codice senza bug della reentrancy

Un altro esempio di codice di un contratto Solidity che possiede lo stesso bug della reentrancy:

```
1  contract SendBalance {
2          mapping (address=>uint) userBalances;
3          bool withdrawn=false;
4
5          function getBalance(address u) constant returns(uint) {
6                  return userBalances[u];
7          }
8
9          function addToBalance() {
10                 userBalances[msg.sender]+=msg.value;
11         }
12
13         function withdraw Balance() {
14                 if (!(msg.sender.call.value(userBalances[msg.sender])()))
15                         throw;
16                 userBalances[msg.sender]=0;
17         }
18 }
```

Evidentemente se l'ultima funzione fosse fatta in questo modo:

```
1
2          function withdraw Balance() {
3                  userBalances[msg.sender]=0;
4                  if (!(msg.sender.call.value(userBalances[msg.sender])()))
5                          throw;
6          }
```

Allora pur contenenendo qualsiasi tipo di codice il contratto di "msg.sender" non potrebbe comunque sfruttare la reentrancy sbagliata e richiamare il contratto "SendBalance", in quanto il suo "userBalances" sarebbe già a 0, impendendogli quindi, nonostante i possibili tentativi di fare double spend e innestare altre chiamate per fare triple spend ed etc.

Altro Esempio di funzione corretta di withdraw per evitare Race-To-Empty:

```
1  function withdrawBalance() {
2          amountToWithdraw=userBalances[msg.sender];
3          userBalances[msg.sender]=0;
4          if (amountToWithdraw>0) {
5                  if (!(msg.sender.send(amountToWithdraw)))
6                          throw;
7          }
8  }
```

Ancora meglio con un meccanismo di mutex:

```
1  function withdrawBalance() {
2          if (withdrawMutex[msg.sender]==true)
3                  throw;
4          withdrawMutex[msg.sender]=true;
5          amountToWithdraw=userBalances[msg.sender];
6          if (amountToWithdraw>0) {
7                  if (!(msg.sender.send(amountToWithdraw)))
8                          throw;
9          }
10         userBalances[msg.sender]=0;
11         withdrawMutex[msg.sender]=false;
12 }
```

Riassumendo: per la reentrancy si può fare l'aggiornamento di una saldo prima di una send e non dopo, evitare la chiamata a un contratto non definito o non conosciuto, e in generale usare un costrutto che renda più atomica possibile la sezione contenente send e aggiornamento del saldo, oppure usare metodi come mutex, lock e sistemi di permessi con livelli di priorità visto che, nonostante i contratti e la rete ethereum non prevedano concorrenza, di fatto questa è un po' come se ci fosse [11]. Oltre la reentrancy infatti sono possibili errori relativi a race-conditions e a non determinismo generato dal succedere di operazioni non previste e cambiamenti di stato non ritenuti possibili dati dall'esecuzione di contratti durante l'esecuzione di stessi.

Si sono provati a fare degli analizzatori sintattici, alcuni presentati in paper inclusi in bibliografia [12][13][14], ma di nessuno di questi è perfetto e nonostante il rilevamento di contratti vulnerabili, non è dato per certo il funzionamento per tutti, o per potenzialmente "nuovi" scenari di reentrancy e race-condition.
Sarebbe efficace se Ethereum stesso sviluppasse un analizzatore sintattico ufficiale, chiaramente open source, e, nel caso, che questo fosse testato a fondo e per lungo tempo. In effetti questo progetto è presente nella roadmap delle grosse patch future, cioè hard fork, tra i quali il più vicino porta il nome di Metropolis. Per creare un analizzatore sintattico efficiente per quanto riguarda la reentrancy bisogna che sia ben individuato il tipo della funzione, cioè del contratto, e ridurre il più possibile le operazioni fino a individuare poche istruzioni basilari e standard, come ad esempio "update del saldo", "invio", "ricezione", etc. Convertendo efficacemente ogni contratto in questo mini-set di pseudo istruzioni quindi analizzare il loro ordine, cercando ogni tipo di pattern già noto che genera mancanza di reentrancy, e quindi possibili intrusioni e cambiamenti nel mezzo dell'esecuzione dello stesso. Chiaramente anche con un analizzatore sintattico teoricamente "perfetto", che riconosce i casi di mancanza di reentrancy, si potrebbero comunque verificare bug legati ad altri fattori, a volte non meramente identificabili riconoscendo un determinato tipo di funzione relativo all'errore stesso, in quanto non funzionali ma interamente dipendenti da chi scrive o utilizza il contratto, e in alcuni casi anche dal non determinismo.

Per questo, tenendo a mente la reentrancy, andiamo ad analizzare altri difetti e vulnerabilità di Solidity (che in alcuni casi non sono solo di questo ma intrinsecamente di tutti i linguaggi per smart contract Ethereum).

4.2 Gas per la Send e Indirizzi non verificati

4.2.1 King of the Ether Throne

Il contratto King of the Ether Throne [15] è un contratto simile a una lotteria: esiste una cifra per reclamare il trono, chi mette la cifra nel contratto diventa re, e il precedente re riscuote ciò che il nuovo re ha pagato, a quel punto la cifra aumenta di 50% e, se qualcuno vuole reclamare il trono, dovrà pagare facendo perdere il titolo al precedente re, ma donandogli il 150% di quanto questo aveva messo in precendenza (compensation), meno una certa percentuale (1%) che va in fee al gestore del contratto. Instaurato un nuovo re, in 14 giorni qualcuno può reclamare il trono e nel caso nessuno lo faccia, il re verrà aggiunto alla lista di re sulla pagina web del contratto, ma non riavrà mai indietro i soldi versati in questo.

4.2.2 Differenze tra account

Il bug in questione, in questo contratto, è un altro di quelli particolarmente dannosi, e di cui è necessario avere una buona conoscenza e comprensione se si vuole programmare in Solidity. Di fatto esistono due diversi tipi di account all'interno della rete Ethereum: account normali (externally-owned), cioè indirizzi senza nessun contratto associato, e account contratto. Il primo è controllato da una persona fisica, il secondo è relativo a un contratto.
Il browser Mist da la possibilità di creare un account normale, ma incoraggia i propri utenti anche a farsi un "contract-based wallet", per migliorare la sicurezza e avere più opzioni di gestione sull'indirizzo associato a questo, come ad esempio un limite giornaliero sul prelievo, pagamenti automatici, etc. Un qualsiasi pagamento viene iniziato sempre da un account normale e, anche nel caso in cui si utilizzi un account contratto, ci deve essere un account normale dietro di questo.

Il codice del contratto in questione:

```
1  contract KingOfTheEtherThrone {
2          /.../
3          struct Monarch {
4                  address ethAddr; // l'indirizzo del re
5                  string name;
6                  uint claimPrice; // quanto paga al re precedente
7                  uint coronationTimestamp;
8          }
9          Monarch public currentMonarch;
10         function claimThrone(string name) { // richiedi il trono
11                 /.../
12                 //Qui non controlla il valore di ritorno della send
13                 if (currentMonarch.ethAddr!=wizardAddress)
14                         currentMonarch.ethAddr.send(compensation);
15                 /.../
16                 //Assegna il nuovo re
17                 currentMonarch=Monarch(
18                         msg.sender, name, valuePaid, block.timestamp);
19         }
20         /.../
21 }
```

Di fatto, il contratto non presentava bug nel caso in cui la funzione "compensation()" *(riga 14 del codice del contratto)* era eseguita a favore di un account normale, ma solo nel caso di un indirizzo collegato a un account contratto. Di default, quando un qualsiasi contratto in Ethereum, o la console di un client, effettua una send (o anche un call), viene posto su questa chiamata in automatico un limite massimo di 2300 unità di gas (gasLimit). Questo si è rivelato essere troppo poco nel caso di molte chiamate ad account contratto. Non si riuscivano a pagare le operazioni necessarie al "contract-based wallet" di Mist o a qualsiasi altro per finire la sua esecuzione. La chiamata send quindi lanciava un eccezione "out-of-gas" che risultava in un throw, ma per via del fatto che il contratto King of Ether Throne non controllava il valore di ritorno della send, e per il fatto che questa è un tipo di chiamata indiretta, come spiegheremo più avanti, questo throw aveva effetto solo sulla chiamata fatta dalla send, ma non sul contratto che l'aveva chiamata che, nonostante il fallimento e il ritornare "false" della send, concludeva normalmente la sua esecuzione, noncurante se questa fosse stata eseguita correttamente o meno.

In questo caso il gestore del contratto non era nè al corrente che ci fosse questo gasLimit automatico per la send, nè che questa quantità bastasse per un account normale, o per uno contract con una funzione di fallback triviale, che al massimo produceva un semplice log della transazione ma non per altri che compievano operazioni aggiuntive, come un qualsiasi contratto wallet standard compresa la DAPP wallet, la più usata del browser Mist, nonchè consigliata ufficialmente da Ethereum [16].

Esistono tool, ad esempio etherscan.io [17], per controllare, tra le altre cose, anche se un certo indirizzo è collegato a un contratto e quale indirizzo ha fatto il deploy del contratto su quel determinato indirizzo, e anche metodi con operazioni bytecode che illustreremo più avanti.

4.2.3 La funzione di fallback

Quando si chiama una funzione di un contratto, la chiamata invia una signature che corrisponde a quella della funzione che vogliamo chiamare, nel caso in cui la signature non corrisponda a quella di nessuna funzione o sia semplicemente vuota, cioè il caso della send (ma anche della call e delegatedcall), allora la chiamata fa sì che venga eseguita la funzione di fallback indicata con "function()" e implementata nel contratto [18]. Nel concreto, usando il metodo <address>.send(<amount>) la EVM assegna un limite di 2300 gas alla chiamata, una quantità esigua che basta per ricevere la transazione e che andrebbe usata solo nel caso di un account normale.
Ogni pagamento o trasferimento di denaro in Ethereum è sempre fatto nella forma di una transazione, cioè una chiamata *(send/call/delegatedcall)* con signature vuota da un contratto, o via console di un client, da un indirizzo mittente a un altro indirizzo destinatario il quale può essere normale e ricevere la transazione senza fare operazioni, o collegato a un contratto, e quindi lanciare l'esecuzione della propria funzione di fallback.

Inoltre, non avendo posto un controllo sul valore di ritorno della send, il fallimento di questa non aveva sortito effetti all'interno del contratto principale, tant'è che nemmeno si era a conoscenza del fatto che questa fosse fallita.

4.2.4 Chiamata diretta e indiretta

Per spiegare meglio il motivo per cui il fallimento della send non abbia bloccato l'esecuzione del contratto, e quindi le ragioni per cui è importante controllare il valore di ritorno ed eventualmente implementare manualmente un throw nel caso di fallimento di questa, è necessario evidenziare la differenza tra le chiamate dirette e quelle indirette per invocare (funzioni di) altri contratti:

La chiamata indiretta è quella effettuata da una send/call/delegatedcall a un certo indirizzo. C'è da notare che in EVM non esiste una differenza tra "send" e "call" Solidity, in quanto in entrambi si tratta di agglomerati di operazioni bytecode in cui la vera e propria chiamata è eseguita dall'opcode CALL (o CALLCODE), tant'è che "<addr>.send(amount)" non è che zucchero sintattico per "<addr>.call.gas(0).value(amount)()". Usando una di queste 3 funzioni Solidity di fatto si chiama la funzione di fallback, nel caso in cui l'indirizzo ricevente sia un account contratto chiaramente e quindi ne possegga una.

L'altro tipo di chiamata è quella diretta che invece di usare una send/call/-delegatedcall verso un indirizzo di un contratto, semplicemente usa il nome del contratto e il metodo che vogliamo chiamare "<contratto>.<nomefunzione>()" come una normale chiamata a un metodo pubblico da una funzione appartenente a una classe a una appartenente a un'altra, in un qualsiasi linguaggio ad oggetti simile a Java.

La differenza tra le due, per quanto riguarda le eccezioni, è notevole, di fatto se si verifica un'eccezione durante una chiamata diretta il fallimento si propaga e il throw di una chiamata diretta innestata fa rollback di tutto ciò che è stato eseguito anche dalle chiamate precedenti, tutto il gas assegnato viene consumato e l'esecuzione si ferma.

Al contrario invece, una chiamata indiretta fa rollback solo di se stessa, indi per cui se da una funzione viene fatta una call e all'interno di questa viene sollevata un eccezione, viene fatto il rollback di solo ciò che è stato eseguito a partire dalla call e non di ciò che è stato fatto prima, conseguentemente viene consumato tutto il gas che era stato allocato per questa, a meno che non fosse stato stabilito un upper bound, e questa ritorna il valore "false" mentre l'esecuzione continua con la riga successiva la call nella funzione che l'aveva chiamata.

4.2.5 Cause del bug

Ciò che scatenò il bug in questo caso furono i contratti creati dalla dapp wallet standard di Mist (meteor-dapp-wallet) che riceveva la transazione eseguiva la funzione di fallback contenente varie operazioni fino a un certo punto per poi rimanere senza gas "in itinere", ciò risultava in un rollback di tutte le operazioni bytecode eseguite dalla chiamata fino a quel momento, non essendo però controllato il valore di ritorno della send il contratto King of the Ether Throne continuava la sua esecuzione ignorando il valore ritornato "false" e quindi automaticamente supponendo che la send fosse andata a buon fine, segnando come pagato il re uscente e aggiornando il trono ed il prezzo per reclamarlo.

Le cause di questo bug sono da ricercare, oltre che nell' inesperienza dello sviluppatore del contratto, anche nella generale mancanza di conoscenza riguardo

Solidity stesso, lo stesso sito ufficiale non riporta e menziona in poche righe possibili bug relativi al gasLimit della send, mentre molti esempi presenti online presentano pattern sbagliati che possono generare problemi di questo tipo, ad esempio l'uso di "<address>.send(<amount>)" senza controllare il valore di ritorno. Altro problema nasce dalla consapevolezza che non sia possibile aggiungere del gas extra nella send usando la call.
La funzione di fallback dei contratti wallet "Ethereum Mist Wallet" richiede sicuramente più di 2300 unità di gas per ricevere una transazione, che supera la quantità limite disponibile durante una chiamata "<address>.send(<amount>)".
Questo issue relativo alle funzioni di fallback è noto, genera confusione per molti sviluppatori ed è segnalato sulla pagina github dedicata al browser Mist. Inoltre se è poca la documentazione (corretta) su Solidity è praticamente assente quella sulla EVM, che si limita praticamente al solo "Ethereum Yellow Paper"[4], paper ufficiale che spiega, con un linguaggio fortemente accademico e strettamente matematico, la logica di ogni istruzione della virtual machine ma dal quale è estremamente difficile per uno sviluppatore senza una forte esperienza riguardo macchine virtuali e bytecode ricavare informazioni utili allo sviluppo di un contratto usando Solidity.

4.2.6 Esempi di come chiamare un contratto

"EXTCODESIZE" è un opcode che si riferisce a un' operazione bytecode EVM e ritorna la dimensione del codice collegato a un address, se la dimensione è >0 allora sicuramente l'address è quello di un contratto.
C'è bisogno per poter utilizzare questo opcode di scrivere codice assembly visto che il compilatore Solidity ancora non lo supporta direttamente e non esiste una funzione di alto livello, al pari delle send/call per l'opcode CALL, che lo utilizza.

Un esempio di come può essere un controllo su un indirizzo:

```
1  function checkIfContract(address _addr) returns(bool _contratto) {
2          uint32 size;
3          assembly {
4                  size:=extcodesize(_addr)
5          }
6          if (size>0)
7                  return true;
8          else
9                  return false;
10 }
```

Una versione "migliorata" della send presente in King of The Ether Throne:

```
1  function trasferireSoldi(address _destinatario, uint _amount)
2  returns(bool _successo){
3          if (!(isContract(_destinatario)))
4          { //entra qui se _destinatario non è un contratto
5                  if (!(_destinatario.send(_amount)))
6                          throw;
7                  return true;
8          }
9          else { //se invece lo è entra qui
10                 if (!(_destinatario.call.value(_amount).gas(100000)))
11                         throw;
12                 return true;
13         }
14 }
```

Il valore 100000 è puramente arbitrario e scelto a caso, si potrebbe ad esempio calcolarlo se si possedesse una funzione "costoDiGas(address addr) returns(uint32)" che dato un indirizzo che punta a un contratto fosse capace, senza doverlo eseguire, di ritornare un approssimazione, magari per eccesso, di quanto gas questo contratto si pensa che richiederà per essere chiamato con successo. Si potrebbe anche aggiungere un ulteriore controllo all'interno dell'else facendo in modo che nel caso in cui venga individuato un contratto di tipo standard, come ad esempio quello wallet di Mist, questo venga riconosciuto più facilmente e sia possibilmente meno dispendioso calcolare il gas che si presume venga usato durante la chiamata.

Come best practice quindi troviamo:

- Evitare di usare costrutti <address>.send(<amount>) a meno di non essere sicuri che l' <address>sia un address normale (externally-owned) o che sia un contratto la cui funzione di fallback può necessitare di 2300 gas e non oltre.

- Considerare l'idea di usare throw per far ritornare i fondi al chiamante rispetto che un'altra send/call.

- Usare un costrutto tipo <address>.call.value(value).gas(extraGasAmt)() per la chiamata a un contratto qualsiasi, anche se in questo caso potrebbe risultare difficile calcolare un "extraGasAmt" che sia alto abbastanza da risultare sufficente per la maggior parte dei contratti che ricevono e al contempo abbastanza basso per essere nella disponibilità della maggior parte di coloro che vogliono chiamare il contratto.

- Esaminare sempre il valore di ritorno di send() e call() e agire in base a questo valore. A volte lanciare un throw dopo un errore può essere la scelta migliore a volte invece può essere meglio effettuare altre operazioni e non interrompere l' esecuzione con un rollback totale.

- Fare attenzione ai contratti "poison". Un contratto poison è un contratto pensato unicamente per fini distruttivi e costituisce, se il contratto che lo chiama non gestisce la situazione a dovere, un possibile deadlock e starvation per tutti coloro che si trovano ad interagire col contratto in questione similmente a quello che farebbe un attacco DoS.

- Sviluppare o cercare tool migliori per l'analisi della blockchain che possano individuare chiamate da contratto a contratto e nel caso di fallimento salvare l'address e notificare che il pagamento non è effettivamente arrivato.

- Conoscere molto bene e a fondo, per quanto difficile, la EVM soprattutto per le questioni riguardanti il gas.

- Effettuare molti test.

4.3 Poison contract

4.3.1 L'obiettivo

Inviare soldi a un address dinamico, che può essere collegato sia a un account normale o a uno contratto, è sempre difficile in quanto non sanno preventivamente le operazioni che questo contratto che chiamiamo debba eseguire e quindi quale limite di gas sarebbe opportuno stabilire al momento della chiamata. Un contratto poison è un contratto che viene costruito solo a fini distruttivi, non "ruba" soldi e non genera introiti per l'attaccante, si tratta invece di un disservizio simile a quello generato da un attacco DoS se il contratto attaccato ne è vulnerabile. Un contratto di questo tipo è scritto in modo da usare una quantità così esorbitante di gas per concludere la propria esecuzione quando chiamato, che praticamente nessuno riuscirà, fatta la chiamata a non interrompersi a metà: ogni chiamata fallirà sistematicamente in un'eccezione out-of-gas, ritornando "false" come di default per ogni chiamata Solidity che genera un eccezione, che come sempre non viene raccolta e risulta in un throw [19]. In un contratto in cui si controlla il valore di ritorno della send/call e la si riprova, magari in forma di una "<address>.call.value(<amount>).gas(<gas aggiuntivo>)" neanche con un valore esagerato di gas aggiuntivo (a partire da >= 2300) la chiamata potrà mai andare a buon fine ed è quindi impossibile fare una chiamata che non ritorni falso e che non si interrompa prima di finire.

4.3.2 I potenziali danni

Qualora poi un contratto dovesse riprovare a intervalli regolari a effettuare una send/call fallita, magari aumentando di volta in volta il quantitativo di gas, finirebbe ineluttabilmente per essere svuotato di tutti gli ETH nel suo saldo se non implementata, in questo, un adeguata misura preventiva. Nel caso in cui invece il contratto fosse scritto in modo tale da non poter proseguire finchè una send/call non fosse stata effettuata con successo, cioè conclusa e con valore di ritorno "true", ciò comporterebbe un blocco simile a un deadlock in programmazione concorrente e conseguente starvation per tutti gli altri utenti che interagiscono col contratto da quel punto in poi senza possibilità di sbloccarlo. Solo l' intervento diretto umano del proprietario potrebbe essere utile, ma solo nel caso in cui procedure di gestione e modifica fossero già state implementate nel contratto stesso, questo per via dell'immutabilità di un contratto che, se non implementato un metodo di selfdestruct, neanche può essere cancellato da colui che ne ha fatto il deploy.

```
1  contract SendContract {
2      address public richest;
3      uint public mostSent;
4      function SendContract() payable {
5          richest=msg.sender;
6          mostSent=msg.value;
7      }
8      function becomeRichest() payable returns(bool) {
9          if (msg.value>mostSent) {
10             richest.transfer(msg.value);
11             richest=msg.sender;
12             mostSent=msg.value;
13             return true;
14         } else {
15             return false;
16         }
17     }
18 }
```

Questo contratto, preso dalla pagina ufficiale della documentazione di Solidity [20], mostra come un implementazione del genere di una funzione di withdraw ispirata al contratto King of the Ether Throne sia suscettibile a deadlock se attaccata da un poison contract. In questo caso è sufficente che "richiest" sia un account contratto e che l'esecuzione della sua funzione di fallback costi più di 2300 unità di gas, questo farà sempre fallire la riga
"richest.transfer(msg.value)" e di conseguenza la funzione
"becomeRichest()" rendendo impossibile a chiunque fare qualsiasi tipo di operazione col contratto bloccato per sempre.

4.3.3 Come proteggersi

```
1  contract WithdrawalContract {
2      address public richest;
3      uint public mostSent;
4      mapping (address => uint) pendingWithdrawals;
5
6      function WithdrawalContract() payable {
7          richest=msg.sender;
8          mostSent=msg.value;
9      }
10
11     function becomeRichest() payable returns(bool) {
12         if (msg.value>mostSent) {
13             pendingWithdrawals[richest]+=msg.value;
14             richest=msg.sender;
15             mostSent=msg.value;
16             return true;
17         } else {
18             return false;
19         }
20     }
21
22     function withdraw() {
23         uint amount = pendingWithdrawals[msg.sender];
24         pendingWithdrawals[msg.sender]=0;
25         msg.sender.transfer(amount);
26     }
27 }
```

La sua variante migliorata invece, fa si che l'attaccante e possessore del contratto poison sia in grado solamente di far fallire il suo withdraw ma non quello degli altri e non di bloccare l'esecuzione generale del contratto.
Qui se il contratto in questione è poison, quindi impossibile da eseguire, il danno ricade solamente su colui che lo utilizza in quanto il suo indirizzo "msg.sender", richiedendo troppo gas per eseguire la funzione di fallback, fa fallire l'ultima riga della funzione "withdraw()" cioè "msg.sender.transfer(amount)". In questo modo il trasferimento dei suoi fondi verso il suo indirizzo fallisce ma essendo i fondi da ritirare degli utenti organizzati in un array, l' esecuzione prosegue con i successivi partecipanti, dopo aver azzerato il saldo dell'attaccante, giustamente prima della funzione "transfer" così da evitare possibili attacchi con la reentrancy, come osservato al punto 4.1.
Per migliorare l'esempio si potrebbe aggiungere un controllo sul valore di ritorno di "msg.sender.transfer(amount)" e nel caso di esito negativo, associando il fallimento al saldo e all'indirizzo in questione, salvare in un altro array l'informazione che il ritiro del saldo da parte di quell'indirizzo è fallito e comunicarlo al proprietario del contratto in modo che provveda manualmente alla risoluzione del problema, magari contattando in qualche modo l'indirizzo stesso o scrivendolo in qualche pagina web per poi richiedergli un altro indirizzo, questa volta associato a un account normale e non a uno contratto, questo per gestire anche l'evenienza che il tentato attacco poison fosse involontario e più simile al bug riscontrato nel punto 4.2.

Un altra possibile soluzione al problema, sempre proposta dalla documentazione ufficiale di Solidity [20], è quella di filtrare gli indirizzi, metodo efficace e che consente di non doversi preoccupare del fatto che il proprio contratto sia o meno vulnerabile a un possibile disservizio generato da un contratto poison. Un contratto privato già per definizione è un contratto di cui non sono visibili le variabili di stato e che può essere chiamato solamente dagli indirizzi presenti nella sua white list. La lista è definita dal contratto stesso e questo esclude a priori la possibilità di un attacco da parte di un contratto poison, anche nell'evenienza in cui il contratto in questione ne fosse eventualmente vulnerabile. Un contratto pubblico invece è visibile e chiamabile da tutti, è possibile oscurare o cryptare alcuni valori all'interno di esso, ad esempio lo stipendio di un dipendente, tramite meccanismi di chiavi private/pubbliche o altri tipi di crittografia, ma non è invece possibile oscurare il contratto a certi indirizzi, a meno appunto di non usare un sistema di filtraggio. È bene notare che di default le variabili di stato di un contratto sono private e lo rimangono a meno di non dichiararle esplicitamente pubbliche tramite la keyword "public".

Il filtraggio si può basare su numerosi parametri ed è implementato attraverso l'uso di modificatori definiti dalla keyword
"modifier", la struttura di questi è definita come segue:
modifier <nome del modificatore>(<argomenti>)
{require(<condizione>); _; //possibile altro codice seguente}
come una funzione il modificatore ha un nome e degli argomenti, al suo interno si trova una condizione che se soddisfatta sostituisce "_;" con il corpo della funzione all'interno della quale si trova il modificatore, in caso contrario non compie quest'operazione ed esegue solo il codice aggiuntivo, se ne contiene, che viene eseguito in entrambi i casi a prescindere se la condizione è stata rispettata o meno. La condizione di "require" può essere di vari tipi differenti, può essere un controllo su un indirizzo, sul tempo, un confronto tra due parametri ma anche un controllo sullo stato a cui il contratto si trova, se è a un certo stato e a un certo tempo, se una certa operazione è già stata o meno eseguita. Il modificatore può essere anche usato per gestire in modo automatico le transazioni, ad esempio, nell'eventualità di doverne fare più di una in sequenza, si può usare il modificatore per passare all'esecuzione della successiva o anche per fare transazioni temporizzate automatiche.

4.4 Limite dello stack

4.4.1 Evitare lo stack overflow

Un' altra vulnerabilità che è stata sfruttata in più di un occasione è quella del cosidetto attacco allo stack. L' Ethereum Virtual Machine esegue un bytecode composto da circa 100 istruzioni che, come altri linguaggi di programmazione, utilizza uno stack in cui vengono salvati, in forma di record, i dati relativi alla chiamata e all'esecuzione di un contratto e impilati su di esso successivamente i record delle chiamate innestate, senza svuotare lo stack fintanto che la chiamata di partenza non abbia concluso la sua esecuzione. Nel caso di molte chiamate innestate all'interno di essa questo significa che, quando queste hanno terminato, in ordine inverso a quello della loro chiamata, la loro esecuzione, le viene restituito il controllo e la funzione può terminare.

Ogni volta che un contratto o una funzione vengono invocati e la loro chiamata non è stata fatta all'interno di un'altra funzione o di un altro contratto, lo stack è vuoto (fattivamente un contratto non è che una funzione). Nel caso in cui un contratto/funzione effettui numerose chiamate ad altri contratti/funzioni contenenti possibilmente ulteriori chiamate, visto che lo spazio di archiviazione della rete non è infinito, serve un limite massimo per l'altezza della pila. Un altezza massima non definita potrebbe portare, oltre che a possibili sovraccarichi e rallentamenti della rete, al rischio vero e proprio di incorrere in un uno stack overflow, all'esaurirsi dello spazio totale.

4.4.2 Sfruttare il limite

Questo limite, al momento, è di 1024 record, senza stabilirlo l' esecuzione di un contratto scritto male, volontariamente o meno, potrebbe portare sia a casi di starvation per altri contratti, sia a un utilizzo eccessivo dello spazio. Ciò potrebbe ridurre le performance generali della rete e portare, nel caso si usasse tutto lo spazio disponibile, alla possibilità di sovrascrivere l'heap. Solidity è l'unico tra i linguaggi per smart contract Ethereum che esegue un throw automatico in corrispondenza della chiamata che eccede lo stack limit, Serpent ad esempio non effettua questa operazione ed è necessario controllare questo parametro manualmente.

4.4.3 King of the Ether Throne

Il contratto King of The Ether Throne è vulnerabile anche a questo attacco, nella riga
“currentMonarch.ethAddr.send(compensation)”:
dopo aver eseguito 1022 send/call a un qualsiasi altro contratto o funzione, magari anche allo stesso da cui si opera l'attacco, si chiama il contratto King of The Ether Throne, la cui chiamata risulta essere quella che genera il 1023esimo record impilato sullo stack. La send quindi, essendo la prima chiamata eseguita dopo l' invocazione del contratto, risulta essere la 1024esima e fallisce lanciando in automatico un eccezione che, non raccolta in Solidity, genera un throw con effetto

di rollback di tutto ciò che è stato eseguito dall'ultima chiamata effettuata, cioè quest'ultima. Questa send avrebbe pagato il "currentMonarch", ma fallisce come se, per qualche motivo, fosse stata sollevata un eccezione out-of-gas o fosse incorsa in un throw implementato nel contratto stesso, riuscendo così a non far arrivare i soldi a colui al quale spettavano. L'attacco risiede nello sfruttare per scopi di disservizio questo meccanismo, ad esempio da un contratto del genere si potrebbe spodestare il re senza che questo venga pagato. Non si "ruba" niente nel caso di King of The Ether Throne, a meno che l'attacco non sia ad opera del proprietario stesso del contratto, come è successo in altri casi tra cui il contratto EtherPot, una lotteria online a premi. Di fatto osservando le chiamate che un contratto effettua si possono eseguire un certo numero di chiamate "vuote" prima dell'invocazione del suddetto contratto vulnerabile e si può così far fallire l'esecuzione di una specifica chiamata all'interno di esso con effetti anche disastrosi nel caso in cui il contratto non sia stato implementato a dover per far fronte a questa vulnerabilità. Una vincita maggiore o maggiori interessi per l'attaccante nel caso di alcuni contratti contenenti schemi di Ponzi/a piramide, e in altri simili, una dinamica analoga a quella vista in The DAO se il contratto aggiorna il nostro saldo tramite una chiamata (molto sconsigliato) posta dopo la send/call di pagamento verso il nostro contratto, facendo un numero opportuno di chiamate prima dell'invocazione, fallirebbe la chiamata per l'aggiornamento del saldo, mentre la send di pagamento verrebbe eseguita con successo.

4.4.4 GovernMental

Un altro contratto che presenta questa stessa vulnerabilità è "GovernMental" che approfondiremo nel punto 4.5 anche per altri difetti.
Allo stesso modo di King of The Ether Throne questo contratto non controlla il valore di ritorno della send e nel caso in cui il proprietario invochi un contratto come il seguente cade nelle stesse dinamiche.

```
1 contract attaccoDelProprietario {
2         function attacca (address bersaglio, uint count) {
3                 if (count>=0 && count<1023)
4                         this.attacca.gas(msg.gas-2000)(bersaglio, count+1);
5                 else
6                         Governmental(bersaglio).resetInvestment();
7         }
8 }
```

La funzione del contratto che contiene la vulnerabilità:

```
1  function resetInvestment() {
2          if (block.timestamp<lastInvestmentTimestamp+ONE_MINUTE)
3                  throw;
4
5          lastInvestor.send(jackpot);  //Questa è la riga che fallisce
6          owner.send(this.balance-1 ether);
7
8          lastInvestor=0;
9          jackpot=1 ether;
10         lastInvestmentTimestamp=0;
11 }
```

Essendo la seconda send diretta al proprietario e contenente tutto il balance, cioè il saldo del contratto, ed essendocene però una precedente che manda "jackpot", cioè la metà, a "lastInvestor", è facile capire perché sia conveniente per il proprietario "owner" che la prima send fallisca mentre che la seconda vada a buon fine senza registrare il fallimento della prima. Di fatto l'esecuzione della funzione "resetInvestment" inizia con uno stack già pieno per 1022 record, di cui quello associato alla chiamata di "resetInvestment" è il 1023esimo creato, giunti quindi alla prima chiamata all'interno di questa funzione, cioè "lastInvestor.send(jackpot)", questa risulta essere la 1024esima e fallisce per spazio esaurito sullo stack. Non essendo poi controllato il valore di ritorno di questa l'esecuzione non si interrompe e prosegue con la send nella riga successiva che fallisce anch'essa visto lo stack ancora pieno per 1023 record alla stessa maniera della precedente. Invocando una seconda volta però la funzione con stack vuoto, fattivamente non ci dovrebbe essere più nessun investitore da pagare e il contratto dovrebbe avere il balance a zero. Non essendoci però più un "lastInvestor" il contratto esegue di fatto solamente la seconda send mandando tutto il balance al proprietario.

4.4.5 Rimedi e il fix

Un modo per evitare scenari di questo tipo, oltre alle best practice già citate ovvero di aggiornare il saldo prima di inviare denaro o di fare in modo che un throw, in caso di fallimento di una delle due, faccia rollback delle operazioni fatte da entrambe, è quello di controllare lo stato dello stack all'inizio dell'esecuzione di un contratto e ad esempio rifiutare le chiamate nel caso in cui lo stack sia quasi pieno o non vuoto abbastanza da permettere un esecuzione del contratto chiamato senza interruzioni. In maniera più radicale invece, un contratto potrebbe richiedere di non essere mai chiamato come chiamata innestata e potrebbe rifiutare a prescindere chiamate da altri contratti o funzioni, accettando solo chiamate "pulite" o singole, oppure forzando lo svuotamento dello stack come prima operazione.

È doveroso aggiungere che da un anno a questa parte questa vulnerabilità pare essere stata risolta da un hard fork proposto dal creatore della piattaforma Ethereum, Vitalik Buterin, dal nome "EIP-150" [21] ; dopo questa modifica il limite della profondità dello stack non è più ben definito da una costante, cioè fattivamente lo è ancora, ma il limite di 1024 è diventato teorico e irraggiungibile nella pratica. Il numero di chiamate innestate a un certo punto si autolimita (attorno alla 340esima chiamata) visto che l'uso dello stack e l'operazione di effettuare chiamate innestate diventano elementi legati a fattori relativi al gas, indi per cui vengono trattati come risorse, e il rischio di chiamate con stack troppo profondi si vanifica similmente a quello di avere attacchi DoS in quanto sono entrambe procedure troppo costose per essere portate a termine nella pratica. Nonostante ciò non è ancora totalmente certo che quest'attacco non sia attuabile in alcune circostanze particolari e che non si possa ripresentare in futuro in una forma diversa, magari trovando vulnerabilità relative a nuovi meccanismi di parent e child. È quindi comunque importante conoscerlo a scopo preventivo per prendere consapevolezza delle altre vulnerabilità nei contratti e, in particolare, del significato della relazione tra bytecode EVM e Solidity.

4.5 Timestamp attack e Random

4.5.1 Difficoltà del random

Un altro tipo di attacco molto pericoloso è quello che sfrutta la vulnerabilità presente in alcuni contratti che fanno affidamento sul Timestamp dell' ultimo blocco della blockchain.
È un fatto risaputo in informatica che generare numeri random è un compito particolarmente difficile, questo perché il random di un computer è comunque sempre qualcosa di deterministico e quindi una combinazione di eventi, operazioni, shift fatti per generare numeri sempre diversi, apparentemente a caso, ma fattivamente dietro ai quali è possibile riconoscere uno schema, un modello o una parvenza di questi.
Nel caso di blockchain e contratti questo problema viene risolto nei due modi che illustreremo e che i quali sono entrambi vulnerabili per certi aspetti intrinseci e/o di implementazione.

4.5.2 theRun

```
1  uint256 constant private salt=block.timestamp;
2  function random(uint Max) constant private returns(uint256 result) {
3      //trova il miglior seed per generare random
4      uint256 x=salt*100 / Max;
5      uint256 y=salt*block.number / (salt\%5);
6      uint256 seed=block.number/3+(salt\%300)+Last_Payout+y;
7      uint256 h=uint256(block.blockhash(seed));
8      //scegli un numero random tra 1 e Max
9      return uint256((h / x))\%Max+1;
10 }
```

Si prenda ora in esame il contratto theRun [24] , di cui sopra è riportato il codice, questo utilizza il primo metodo che andiamo ad analizzare. Si tratta di un contratto lotteria che, basandosi su un numero "random", estrae il vincitore tra i vari "player" che stanno partecipando al gioco e che hanno depositato una cifra a loro scelta al momento della loro entrata, questa poi servirà da moltiplicatore per la loro vincita, nel caso in cui siano scelti come tali dall'esito della lotteria.
La vulnerabilità di questo contratto è detta "dipendenza dal Timestamp" ed è relativa al funzionamento stesso della blockchain.
Quando un miner mina un blocco, cioè esegue il lavoro necessario a generare un proof (of work in questo caso) che lo rende coerente con il precedente e la chain stessa, aggiunge poi al blocco minato un timestamp derivato dal tempo logico della sua macchina e che ha un range di tolleranza di 900 secondi, dopo aver minato il blocco e aggiunto il timestamp del momento in cui è finito il mining, il blocco viene aggiunto alla blockchain e gli altri miner iniziano il processo di verifica, cioè lo accettano come valido e lo aggiungono anche loro alla loro chain e via via esso viene aggiunto e verificato da sempre più miner.
Di fatto questo metodo non genera problemi ma è abbastanza permissivo, per accettare un timestamp i miner che verificano controllano che questo sia superiore al

timestamp dell'ultimo blocco già aggiunto e verificato della chain e che il timestamp di questo nuovo blocco sia all'interno di un range di 900secondi dal tempo locale del loro sistema al momento della verifica. A questo punto se entrambe le condizioni sono vere, allora il blocco viene verificato. Il problema in questo contratto sorge perché questo utilizza il timestamp dell'ultimo blocco minato della rete come valore e, visto che solitamente sarebbe difficile prevedere tale valore, costruisce la sua funzione random rispetto a questo, il "salt" della funzione "random" infatti non è altro che proprio "block.timestamp".
La vulnerabilità di questo metodo, che all'apparenza può sembrare valido, è che un miner che è anche un player di questo contratto può influenzare attivamente l'outcome della funzione random, facendo dei calcoli e scoprendo quanto e come il valore di salt influenzi l'output, cosa fattibile vista la visibilità pubblica del codice. Minando quindi blocchi con timestamp arbitrari e "giocando" con i 900secondi di treshold a sua disposizione, il miner attaccante può influenzare l'esito della lotteria a sua favore o a sfavore di qualcun altro. Un altro scenario possibile è quello in cui il timestamp dell'ultimo blocco minato viene utilizzato come trigger per un determinato evento, anch'esso implementato in un certo modo, può essere soggetto ad attacchi simili a quello a cui è vulnerabile theRun.

4.5.3 GovernMental

Il contratto a schema di Ponzi "GovernMental" [22] [23] funziona come segue: ogni partecipante entra nel contratto dovendo versare la metà dei soldi che questo possiede nel suo saldo, poi, se per 12 ore nessun altro partecipante entra nello stesso, l'ultimo entrato si spartisce il jackpot a metà con il proprietario, il round si definisce chiuso e ne inizia uno nuovo. Per essere considerato un ultimo partecipante legittimo, questo deve esserlo stato per almeno un minuto. Questo minuto però non è di tempo "reale" verificato dall'esterno ma è di tempo calcolato utilizzando la blockchain e le sue transazioni.
Questo contratto è vulnerabile a vari tipi di attacco:

1. Un attacco allo stack, come visto al punto 4.4, perpetrato dal proprietario del contratto per non pagare il vincitore di un round,

2. Un attacco che sfrutta lo "stato imprevedibile", che non approfondiremo, in cui il miner attaccante costruisce il blocco contenente le transazioni al contratto e arbitrariamente le ordina a sua vantaggio, ponendo la sua come ultima, oppure rifiuta nel suo blocco transazioni altrui facendo così risultare la sua come ultima,

3. Un attacco sul timestamp come il precedente contratto theRun: di fatto la dinamica dell'essere ultimo per almeno un minuto può essere alterata e manipolata facilmente da un attaccante miner che può minare un blocco contenente la sua transazione al contratto GovernMental (ed eventualmente anche rifiutare altre transazioni di altri) a distanza di 12 ore dalla ultima prima della sua, ponendo come timestamp arbitrario esattamente 60 secondi dopo il timestamp del precendente blocco sulla blockchain. In tal modo l'attaccante miner diviene sicuramente l'ultimo partecipante scattate le 12 ore

e, non dando nemmeno un secondo di tempo a nessun altro per inserirsi dopo di lui, diventa con certezza matematica l'ultimo partecipante e "vincitore" di quel round.

4.5.4 Soluzioni, metodi alternativi e Oraclize

In casi come questo la best practice maggiormente consigliabile e forse unica è semplicemente quella di NON usare il valore di timestamp come seed deterministico per generare random, in quanto questo ha una bassa entropia ed è facilmente manipolabile.
Al contrario si può usare, per un contratto che necessita di valori random e volendo usare un qualche parametro relativo alla blockchain stessa, l'indice del blocco, il quale è un incremento, sempre di 1, rispetto all' indice del precendente. In tal modo il contratto non è manipolabile e, nonostante ogni nuovo blocco venga prodotto con una intervallo medio di 12 secondi l'uno dall'altro, questo comunque non rappresenta un tempo certo e dipende da tanti fattori relativi a quanti miner stanno minando in quel momento, quante transazioni sono state avviate, con che fee, dal sovraccarico generale della rete e anche da altri fattori magari economici come il lancio di un nuovo token tramite una ICO (vendita iniziale) o un momento particolarmente propizio per la compravendita. È quindi difficile prevedere a un dato tempo X:XX quale sarà l'indice dell'ultimo blocco minato della blockchain ed è impossibile influenzare questo numero in maniera arbitraria, o in maniera dipendente dal tempo in cui questo viene minato, visto che, a prescindere da quando, questo è sempre e comunque un incremento di 1 rispetto all'indice del blocco precedente. Nonostante ciò però, generare random in generale rimane una sfida e all'interno di contratti che eseguono codice critico / finanziario e che gestiscono forti somme di denaro anche una piccola dose di determinismo o un qualsiasi errore o schema nel generare questo seed può portare a conseguenze gravissime. In altri sistemi, non contratti su blockchain, solitamente si utilizza quello che si definisce un "oracolo", cioè invece di porsi il problema di generare random attraverso una funzione pseudo-random di difficile attuazione e che può peccare di bassa entropia o problemi di implementazione, ci si rivolge direttamente a un servizio esterno, si fa una richiesta tramite una qualche forma di API a un servizio web e si utilizza la risposta come random senza curarsi di come questa è stata ottenuta. Esistono diverse forme e modalità per fare ciò, Wolfram Alpha possiede un servizio simile ed altre API cercano di generare questo random osservando un fenomeno difficile da prevedere o da relazionare a qualche altro elemento, ad esempio il grado di umidità in un certo punto specifico della terra o in uno casuale, oppure fenomeni di tipo sociale come il numero dei nati in un certo arco di tempo in una certa regione della terra, etc. Il problema di tutto ciò però è che un contratto non può interrogare un sito web in quanto fattivamente non è connesso alla rete internet ma solo a quella Ethereum, un contratto è chiuso all'interno dell'ecosistema che contiene tutti gli altri contratti, può interrogare loro ma non l'esterno e ciò è dovuto a motivi di sicurezza e onde evitare possibili interferenze, fughe di dati o API maliziose che potrebbero alterare il funzionamento del contratto stesso. Un' altra best practice quindi per generare random sarebbe interrogare una di queste API. L'operazione è possibile grazie ad un applicativo emergente noto come "Oraclize" [11] [25] [26] [27] ormai abbastanza diffuso ed utilizzato, che permette tral'altro di fare qualsiasi tipo di richiesta verso l'esterno a una qualsiasi API, database, servizio web, etc. La validità delle risposte che riporta è data da prove crittografiche e funge da valido intermediario tra un DAPP o un contratto e il mondo esterno che sarebbe altrimen-

ti inaccessibile [28]. In questo senso esistono molti progetti in via di sviluppo, ad esempio “Kyber Network” [29] che si propone come intermediario per qualsiasi tipo di transazione, soprattutto per quelle che non sarebbero possibili semplicemente tra dapp e contratti, rendendo possibile lo scambio di qualsiasi token e l’uso di molteplici di questi, anche non appartenenti allo standard ERC20 come Bitcoin, all’interno di un unico contratto.

4.6 Problemi con il tipo e il cast

4.6.1 Difetti del compilatore

Più che di una vulnerabilità qui si tratta di una vera e propria mancanza e non esistono attacchi attuabili sfruttando questo bug, nonostante ciò questo può essere dannoso e alterare il risultato dell'esecuzione di uno smart contract. Come altri compilatori, quello di Solidity, che converte il codice di alto livello in bytecode eseguibile dalla EVM, esegue al momento della compilazione del codice una forma di controllo sul tipo simile ai controlli fatti dai compilatori dei linguaggi "classici". Questo controllo però è abbastanza approssimativo in quanto si accorge solamente di assegnamenti errati se fatti tra tipi di dati differenti e di chiamate a funzioni con argomenti di un tipo effettuate con valori non dello stesso tipo della definizione. Il problema però, è che non è sufficiente assicurarsi che a una variabile indirizzo vengano assegnati valori coerenti, che a una stringa non siano assegnati dei valori interi e che a un array non siano assegnati valori booleani o viceversa. È possibile infatti che ci siano contratti con assegnamenti incoerenti e potenzialmente dannosi che passano il controllo formale sul tipo senza errori o warning.

```
1  contract Alice { //ha un certo indirizzo
2          function ping() returns(1)
3  }
4  contract Bob { //ha un certo indirizzo, diverso da quello di Alice
5
6          struct Alice { function ping() }; //definizione della struttura
7
8          function pong(address addr) {
9                  Alice(addr).ping();
10         }
11 }
```

Nell'esempio sopra riportato si vedono due contratti Alice e Bob ognuno con una funzione ping e pong, il contratto Alice semplicemente possiede la funzione ping e il contratto Bob la funzione pong che contiene una chiamata alla funzione ping di Alice. Il problema di questo codice che passerebbe i controlli in fase di compilazione, visto che non presenta assegnamenti errati, si basa proprio sulla riga "Alice(addr).ping()".

Per quanto sappiamo dal punto 4.2, negli smart contract Ethereum entra in gioco la funzione di fallback se il contratto che viene chiamato, quello associato all' indirizzo "addr" in questo caso, non possiede una funzione chiamata ping, ciò significa che la chiamata porta una signature che non corrisponde a quella di nessuna funzione appartenente al contratto che è oggetto di chiamata, Alice nel nostro esempio. Possedendo per davvero il contratto all'indirizzo "addr" (e cioè Alice) una tale funzione in questo caso la funzione di fallback non ci interessa, questa sarebbe chiamata in uno scenario in cui, ad esempio, la funzione ping di Alice non esistesse, a quel punto la chiamata presente in Bob non sarebbe che una alla funzione di fallback.

La chiamata "Alice(addr).ping()" possiede certamente una signature, infatti è chiaramente una chiamata diretta (una indiretta invocherebbe solo la funzione di fallback e sarebbe fatta tramite send/call). In una chiamata diretta il chiamante

deve dichiarare la struttura del chiamato, Alice, e poi fare un cast dell' indirizzo a tale struttura. Usando quindi questa scrittura il contratto Bob comunica al compilatore che effettivamente "addr" e Alice hanno la stessa struttura e tutto ciò che il compilatore controlla è se la struttura di Alice definita dentro Bob contenga o meno una funzione ping.

4.6.2 Possibili errori e scenari

ERRORI:

1. "addr" potrebbe non essere l'indirizzo del contratto Alice quindi il cast di "addr" alla struttura di Alice potrebbe non avere senso se queste due non sono uguali tra loro.

2. "addr" potrebbe non solo non essere l'indirizzo del contratto Alice, ma neppure l'indirizzo di nessun contratto.

3. non è detto che la struttura di Alice dichiarata da Bob corrisponda alla sua vera struttura, difatti il compilatore controlla solamente che si stia chiamando una funzione che abbia lo stesso nome rispetto ad una definita nella struttura di Alice e non nel vero contratto.

In presenza di uno di questi errori poi, trattandosi di chiamata diretta, l'esecuzione del contratto non solleverebbe alcuna eccezione a run-time e a quel punto gli esiti possibili di "Alice(addr).ping()" potrebbero essere vari.

SCENARI:

1. se "addr" è un indirizzo inventato o sbagliato che non corrisponde a quello di nessun contratto esistente, la chiamata termina senza eseguire nulla, costando 0 gas e senza ritornare nessun valore.

2. se "addr" corrisponde a un contratto ma questo non possiede nessuna funzione con signature uguale a quella di "ping()" inviata durante la chiamata, viene eseguita "function()" e cioè la funzione di fallback del contratto chiamato, qualsiasi esso sia.

3. se "addr" è un contratto e la signature corrisponde a quella di una delle funzioni di "addr", allora quella funzione viene eseguita. Non è certo però che il contratto all'indirizzo "addr", nonostante esista e possegga una funzione "ping()", sia effettivamente Alice e qualora non lo fosse, l'esecuzione proseguirebbe senza nemmeno saperlo.

4.6.3 Rimedi e soluzioni

È facile per questo "bug" definire quali siano le best practice, è abbastanza chiaro infatti che il compilatore non eserciti un controllo formale avanzato: non si preoccupa e non sa se le chiamate tra contratti hanno funzionato nella maniera in cui noi si sperava venisse fatto. In futuro è possibile che questo bug venga risolto da una delle future revisioni (come la EIP-150) o dall' avvento del *grosso* hard fork Metropolis, o magari da uno successivo o dall'introduzione del controllo formale avanzato già progettato nella roadmap dei cambiamenti futuri di Ethereum.

Prima di questo momento però è necessario avere bene a mente il fatto che ogni tipo di controllo sui dati in una chiamata diretta è lasciato all'utente ed è lasciato all'utente anche il compito di gestire gli errori e in primo luogo di identificarli.

Sicuramente sarebbe facile evitare gli errori essendo il proprietario dei due contratti che si vogliono far comunicare, infatti si potrebbe, ad esempio, nonostante nulla sia tecnicamente segreto all'interno della blockchain visto che il codice dei contratti è pubblico, come prima operazione mandare e ricevere un valore "segreto" noto solo all'altro contratto (ma anche a chiunque ne visualizzi il codice) a modo di identificazione per evitare lo scenario 3. Questo sarebbe efficace come metodo solo però nel caso in cui nessuno tenti di effettuare un attacco usando un contratto con una funzione con lo stesso nome e che alla stessa maniera invia questo codice. Allo stesso modo si potrebbero evitare gli altri errori definendo correttamente la struttura di uno all'interno dell'altro e conoscendo e verificando l'indirizzo, questo vanificherebbe anche la possibilità di una attacco in quanto un indirizzo non può essere copiato ne si può fare il deploy di un contratto se già ce n'è uno.

Nel caso contrario, in cui si ha il controllo solo di uno dei due contratti che devono comunicare tra loro, si potrebbe, non potendo gestire direttamente l'altro contratto, comunque implementare per ogni chiamata e operazione controlli e valori di ritorno legati ad alcune condition in modo da gestire o perlomeno sapere in quale scenario ci si trova e quale esito ha avuto la chiamata.

Ad esempio per lo scenario 1 si potrebbe controllare innanzitutto che l'indirizzo che stiamo per chiamare sia davvero quello di un contratto con l'operazione già vista al punto 4.2.6 "extcodesize(addr)" e nel caso, far ritornare alla funzione una stringa contenente la descrizione dell'errore avvenuto prima di eseguire un throw che faccia roll-back di tutte le operazioni, o magari, a discrezione del contratto, continuare l'esecuzione dopo la segnalazione gestendo l'errore o continuando con le successive operazioni.

Per lo scenario 2 invece non esiste un opcode del bytecode EVM per controllare se un altro contratto possegga o meno una funzione con una certa signature, in questo caso però si potrebbe, conoscendo la quantità di gas utilizzata dalla funzione, tentare di capire dopo l'esecuzione, vedendo quanto ne è stato restituito, se si tratta della funzione che volevamo chiamare o di un'altra restringendo così la ricerca. Nel caso in cui questo controllo passi, il successivo potrebbe vertere su un azione della funzione stessa, ad esempio controllare del valore di ritorno se il tipo è quello che ci aspettiamo o se il valore è coerente con quello che ci aspettiamo debba essere restituito, in linea generale quindi, a seconda del tipo di funzione, fare un controllo postumo alla sua esecuzione sul cambiamento di stato che avrebbe dovuto produrre, andando a controllare ogni aspetto controllabile delle conseguenze per capire col

margine più alto di certezza possibile se la funzione che abbiamo chiamato é quella che deve essere.

Per lo scenario 3 la soluzione è la medesima, anche possedendo un controllo formale sulla signature, ed eventualmente implementandolo, prima di effettuare la chiamata con la verifica della struttura del contratto all'indirizzo che stiamo per chiamare, i nomi delle sue funzioni e quindi calcolare le rispettive signature da questi e accertare se ne esiste una che corrisponde a quella della funzione presente nella chiamata del nostro contratto. Anche in questo caso è necessario il controllo stabilito sopra, più "semantico" che prettamente "sintattico" e anche nel caso non si riesca a definire se la funzione eseguita sia quella corretta, implementare sempre controlli, valori di ritorni esplicativi di quali controlli sono stati passati e quali no.

Capitolo 5

Conclusione

La programmazione di smart contract è un campo molto nuovo e avveniristico, come si può vedere dal materiale usato nella bibliografia è difficile trovare un articolo o una pubblicazione che abbia più di due anni e infatti la maggioranza delle fonti è concentrata tra l'anno scorso e quello corrente. Si tratta anche di un campo in cui il codice è da ritenersi critico come se si stessero programmando sistemi di sicurezza di centrali energetiche o sistemi finanziari di grandi banche e aziende. Questo fatto ne rende attualmente molto difficile l'utilizzo, visti i molteplici aspetti che ancora non si sono approfonditi, la mancanza di uno storico e di esperienze passate o di tempi lunghi per il testing. Nonostante ciò ritengo che in un futuro non troppo lontano in cui le tecnologie legate all'utilizzo di blockchain saranno molto rilevanti e utilizzate quotidianamente su scala mondiale, gli smart contract potranno costruire un importante tassello in questo nuovo mosaico di servizi che operano in maniera decentralizzata e sicura. Per tradurre questo progetto in una attuazione pratica è necessario lentamente e metodicamente compiere piccoli passi verso una sempre più profonda comprensione delle meccaniche sottese e verso il miglioramento del pensiero e della costruzione della struttura intrinseca di un contratto nei suoi elementi costitutivi. Il primo passo verso questo obiettivo è appunto quello di tenersi costantemente aggiornati, conoscendo quelli che sono i costrutti da evitare e quelli che invece sono da preferire in una circostanza o nell'altra. Solo giunti a questo punto è possibile sperimentare e fare prove in nuove direzioni tentando di rendere un po' più conosciuta e accessibile al mondo questa disciplina ad oggi ancora oscura, misteriosa e piena di potenziali rischi.

Bibliografia

[1] - Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system* (Bitcoin whitepaper). 31 Ottobre 2008.

[2] - Andreas M. Antonopoulos. *Mastering Bitcoin 2nd edition*. Giugno 2017. https://github.com/bitcoinbook/bitcoinbook & https://www.bitcoinbook.info/.

[3] - Ethereum Foundation. *Ethereum white paper*. 6 Aprile 2014. https://github.com/ethereum/wiki/wiki/White-Paper.

[4] - Dr. Gavin Wood. *Ethereum: A secure decentralised generalised transaction ledger* (aggiornato dopo la revisione EIP-150). 8 Luglio 2017 (6 Aprile 2014 la prima versione). https://ethereum.github.io/yellowpaper/paper.pdf

[5] - Massimo Bartoletti, Livio Pompianu. *An empirical analysis of smart contracts: platforms, applications, and design patterns*. 18 Marzo 2017

[6] - Nicola Atzei, Massimo Bartoletti, Tiziana Cimoli. *A survey of attacks on Ethereum smart contracts*. 22 Aprile 2017

[7] - TheDAO smart contract: http://etherscan.io/address /0xbb9bc244d798123fde783fcc1c72d3bb8c189413code.

[8] - David Siegel. *Understanding the DAO attack*. 25 Giugno 2016. http://www.coindesk.com/understanding-dao- hack-journalists

[9] - Phil Daian. *Analysis of the DAO exploit*. 18 Giugno 2016. http://hackingdistributed.com/2016/06/18/analysis-of-the-dao-exploit/

[10] - Peter Vessenes. *More Ethereum Attacks: Race-To-Empty is the Real Deal*. 9 Giugno 2016. http://vessenes.com /more-ethereum-attacks-race-to-empty-is-the-real-deal/

[11] - Ilya Sergey, Aquinas Hobor. *A Concurrent Perspective on Smart Contracts*. 17 Febbraio 2017

[12] - Jack Pettersson, Lucius Gregory Meredith. *Notes on the DAO reentrancy bug and behavioral types*. https://docs.google.com/document/d /1sGlObhGhoEizBXC30Ww4h1KHKGkmcy4NiCKitIBqiUg/ editheading=h.gm4egb3ql9ps

[13] - Loi Luu, Duc-Hiep Chu, Hrishi Olickel, Prateek Saxena, Aquinas Hobor. *Making Smart Contracts Smarter*. 24 Ottobre 2016.

[14] - Karthikeyan Bhargavan, Antoine Delignat-Lavaud, Cédric Fournet, Anitha Gollamudi, Georges Gonthier, Nadim Kobeissi, Natalia Kulatova, Aseem Rastogi, Thomas Sibut-Pinote, Nikhil Swamy, Santiago Zanella-Béguelin. *Formal Verification of Smart Contracts*.

24 Ottobre 2016
[15] - KingOfTheEtherThrone smart contract: https://github.com/kieranelby/KingOfTheEtherThrone/blob/v0.4.0 /contracts/KingOfTheEtherThrone.sol
[16] - Sito del creatore di King of The Ether Throne: http://www.kingoftheether.com/postmortem.html
[17] - Il block explorer della blockchain Ethereum. https://etherscan.io/.
[18] - Zikai Alex Wen, Andrew Miller. *Scanning live ethereum contracts for bugs*. 16 Giugno 2016 http://hackingdistributed.com/2016/06/16 /scanning-live-ethereum-contracts-for-bugs/
[19] - Peter Vessenes. *Griefing wallets*. 4 Giugno 2016. http://vessenes.com /ethereum-griefing-wallets-send-w-throw-considered-harmful/
[20] - Ethereum foundation, Solidity sito ufficiale. *Common Patterns*. http://solidity.readthedocs.io/en/develop/common-patterns.html
[21] - Vitalik Buterin. *Hard fork EIP-150*. 30 Aprile 2017. https://github.com/ethereum/EIPs/blob/master/EIPS/eip-150.md
[22] - Contratto GovernMental. source code: https://etherchain.org/account /0xF45717552f12Ef7cb65e95476F217Ea008167Ae3code
[23] - GovernMental pagina ufficiale: http://governmental.github.io /GovernMental/
[24] - Contratto theRun. source code: https://etherscan.io/address /0xcac337492149bdb66b088bf5914bedfbf78ccc18code
[25] - Sito ufficiale Oraclize. http://www.oraclize.it/home
[26] - Area developer, usare le API e scrivere contratti. https://dev.oraclize.it/
[27] - Oraclize source code: https://github.com/oraclize/docs/wiki
[28] - *Oraclize – Smart Blockchain Oracle Service For Data-Rich Contracts?* https://bitcoinexchangeguide.com/oraclize/
[29] - Kyber sito ufficiale: https://kyber.network/
[30] - Ethereum Foundation. *The solidity contract-oriented programming language*. https://github.com/ethereum/solidity.
[31] - Kevin Delmolino, Mitchell Arnett, Andrew Miller, Ahmed Kosba, Elaine Shi. *Step by Step Towards Creating a Safe Smart Contract: Lessons and Insights from a Cryptocurrency Lab*. 18 Novembre 2015
[32] - *Ethereum Contract Security Techniques and Tips*. https://github.com/ConsenSys/smart-contract-best-practicesdos-with-block-gas-limit
[33] - Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, Eric Koskinen. *Adding Concurrency to Smart Contracts*. 15 Febbraio 2017