PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

SCRIPTING BITCOIN

2DUP EQUAL NOT VERIFY SHA256 SWAP SHA256 EQUAL

Gli script bitcoin lavorano con una struttura di tipo stack, se un utente fornisce la soluzione allo script, la transazione verrà sbloccata.

- ***DUP*** sta per duplication, quindi prendo i primi due elementi dello stack, si duplicano, e li rimetto nello stack. Quindi se nello stack avevo la stringa A B, ora ho A B A B.
- ***EQUAL*** controlla se gli operandi sono uguali, quindi confronto A e B che si trovano in cima allo stack, ottengo false, e quindi metto false in cima allo stack.
- ***NOT*** inverte il valore booleano in cima allo stack. In cima allo stack c'è false, quindi viene preso e invertito, diventa true, e viene messo in cima allo stack.
- ***VERIFY*** controlla se sul top dello stack si trova true. Se non ci sta, invalida la transazione. Adesso sullo stack ci sono A e B. Ho fatto la duplicazione perché prima ho controllato se le stringhe fossero diverse, e i valori di A e B sullo stack sono stati "consumati". Prendo quindi B, calcolo
- ***SHA256*** e il risultato lo metto nel top dello stack al posto di B. A questo punto sullo stack ci sta A sotto e SHA256 di B sopra.
- ***SWAP*** scambia questi due valori, togliendoli dallo stack e inserendoli in ordine opposto. A questo punto in cima allo stack c'è A.
- Calcolo ***SHA256*** di A e lo metto sullo stack.
- ***EQUAL*** controlla se SHA256 di A e SHA256 di B sono uguali. Se è true, la transazione è sbloccata.

Questo script ha lo scopo di sbloccare la transazione se l'utente fornisce due stringhe la cui firma SHA256 è uguale.

DUP HASH160 62e907b15cbf27d5425399ebf6f0fb50ebb88f18 EQUALVERIFY CHECKSIG

Supponiamo di avere un elemento nello stack, ad esempio A, lo script funzione nel modo seguente:

- ***DUP***, prende un elemento dallo stack, in questo caso A, e lo duplica, inserendo il duplicato in cima allo stack. In questo caso abbiamo nello stack: A A;
- ***HASH160***, effettua il digest dell'elemento in cima allo stack, in questo caso si fa il pop del valore A ed effettua la funzione hash su A che ritorna una stringa di 160 bit e la inserisce in cima allo stack. In questo caso abbiamo nello stack: A Hash256(A);
- ***62e907b15cbf27d5425399ebf6f0fb50ebb88f18***, questa è una stringa di dati esplicita, essa viene inserita in cima allo stack effettuando quindi un push. Lo stato dello stack sarà: A Hash160(A) 62e907b15cbf27d5425399ebf6f0fb50ebb88f18;
- ***EQUALVERIFY***, controlla se le due stringhe in cima allo stack sono uguali, ritornando un valore booleano, e contrassegna la transazione come non valida se questo valore è falso. Lo stato dello stack sarà: A;
- ***CHECKSIG***, esegue il controllo della firma della transazione risolvendo così l'accesso.

Questo script ha lo scopo di sbloccare la transazione se l'hash160 di una stringa fornito dall'utente è uguale a 62e907b15cbf27d5425399ebf6f0fb50ebb88f18.

---

2 3 ADD 6 EQUAL

- Viene inizialmente eseguito un push di 2 e successivamente un push di 3 all'interno dello stack. Lo stato dello stack sarà: 2 3 (con 3 in cime allo stack);
- ***ADD***, si effettua un addizione tra i primi due elementi in cima allo stack che vengono anche "consumati", in questo caso si fa 2 ADD 3 = 5. Lo stato dello stack sarà: 5;
- Viene effettuato un push del valore 6. Lo stato dello stack sarà: 5 6;
- ***EQUAL***, controlla se i primi due elementi in cima allo stack sono uguali, se sono uguali aggiunge true in cima allo stack altrimenti false. Lo stato dello stack sarà: 5 6 false;

Questo script non viene mai sbloccato, siccome controlla se 5 è uguale a 6.

CHAINCODE IN GO

```go
package main
import "fmt"

func ping(pings chan<- string, msg string) {
    pings <- msg
}
func pong(pings <-chan string, pongs chan<- string){
    msg := <-pings
    pongs <- msg
}
func main() {
    pings := make(chan string, 1)
    pongs := make(chan string, 1)
    ping(pings, "passed message")
    pong(pings, pongs)
    fmt.Println(<-pongs)
}
```

La dichiarazione ***package*** indica che il programma appartiene al package "main". La parola chiave ***import*** indica le librerie che si intende importare, esse permettono di utilizzare le funzionalità descritti in tali package, in questo caso "fmt" è una libreria standard per input/output. La ***funzione ping*** prende come parametri un canale, utilizzato per il solo invio di stringhe (se si tenta di ricevere un valore da questo canale viene generato un errore di compile-time) chiamato pings e una stringa chiamata msg. Questa funzione non fa altro che inviare la stringa msg al canale pings. La ***funzione pong*** prende come parametri un canale, utilizzato per la sola ricezione di stringhe, chiamato pings e un altro canale, utilizzato per il solo invio di stringhe, chiamato pongs. La funzione dichiara una variabile msg che riceve una stringa dal canale pings, successivamente, questo msg viene inviato all'altro canale pongs.

La ***funzione main*** crea due canali utilizzando la funzione make, che specifica il tipo di valore che possono veicola e la dimensione del buffer, in questo caso è 1. In seguito, chiama la funzione ping passando come argomento il canale pings e il messaggio che si intende inviare su quel canale, ovvero "passed message". Successivamente, si chiama la funzione pongs passando come argomento i due canali, pings e pongs. Infine, viene eseguita una stampa del valore presente nel canale pongs, in tal caso viene stampato msg.

```
package main
import (
    " fmt "
    " time "
)
func worker ( done chan bool ) {
    fmt . Print (" working ... ")
    time . Sleep ( time . Second )
    fmt . Println (" done ")
    done <- true
}
func main () {
    done := make ( chan bool , 1)
    go worker ( done )
    <-done
}
```

Ogni programma in Go ha una dichiarazioni dei package, significa che il programma appartiene al ***package*** "main". L'***import*** prende fmt e time che sono rispettivamente la libreria standard di input/output e la libreria per la gestione del tempo. La ***funzione worker*** prende come argomento un canale di booleani. I ***canali*** sono delle pipe che connettono goroutine concorrenti. Una pipe è uno degli strumenti di comunicazione tra processi offerti dal sistema operativo. Dopodiché stampa "Working...", aspetta 1 secondo e stampa "done". Poi mette il valore true nel canale.
La ***funzione main*** crea il canale done con la funzione make, indicando che è un canale di booleani e che può essere scritto un valore alla volta (il secondo valore opzionale indica la grandezza della coda di bufferizzazione del canale). Passandogli un valore più alto posso far sì che si crei una coda di messaggi che vengono estratte in maniera FIFO per la scrittura sul canale. Dopodiché con la parola chiave ***go*** viene creato un thread e chiamata la funzione worker passandogli il riferimento al canale appena creato.
L'ultima istruzione, <-done, indica di leggere dal canale. È un modo per far sì che il padre (main) aspetti che il figlio termini (worker) per terminare anche lui.

```
package main
import (
    "fmt"
    "time"
)
func main() {
    c1 := make(chan string)
    c2 := make(chan string)
    go func() {
        time.Sleep(1 * time.Second)
        c1 <- "one"
    }()
    go func() {
        time.Sleep(2 * time.Second)
        c2 <- "two"
    }()
    fmt.Println("received", <-c2)
    fmt.Println("received", <-c1)
}
```

La parola chiave ***package*** indica che il programma si trova nel package "main". La parola chiave ***import*** indica che vengono importate delle librerie nel programma, in questo caso "fmt" e "time" sono rispettivamente una libreria standard di input/output e gestione del tempo. La ***funzione main*** vengono dichiarati due canali utilizzando la funzione make che indica come parametro il tipo di valore che possono veicolare, in questo caso string, in più non viene definito una dimensione del buffer e quindi sono canali senza buffer e sono bloccanti, significa che bisogna esplicitare chi riceve su quel canale, altrimenti si ha errore. Successivamente, con la parola chiave ***go***, vengono definite due goroutine che avviano un thread ciascuno ed eseguono una funzione anonima, ovvero una funzione senza nome che è possibile definire dentro ad altre funzioni (in questo caso nel main). La prima funzione anonima attende 1sec e invia al canale c1 la stringa "one", mentre la seconda funzione attende 2sec e invia la stringa "two" al canale c2. Infine, vengono stampati i rispettivi valori che sono stati inviati ai canali c1 e c2. Più precisamente, verrà stampato prima "received two" dopo 2sec circa e "received one" subito dopo, dato che dopo 1sec lo avrebbe stampato ma c'è prima la stampa del valore di c2.
Se non ci fossero state le stampe con <-c2 e <-c1, il programma avrebbe dato errore siccome i canali senza buffer non avevano ricevitori.

SMART CONTRACT IN SOLIDITY

```
pragma solidity ^0.4.21;
contract Coin {
        address public minter ;
        mapping ( address => uint ) public balances ;
        event Sent ( address from , address to , uint amount );

        function Coin () public {
                minter = msg . sender ;
        }
        function mint ( address receiver , uint amount ) public {
                if ( msg . sender != minter ) return ;
                balances [ receiver ] += amount ;
        }
        function send ( address receiver , uint amount ) public {
                if ( balances [ msg. sender ] < amount ) return ;
                balances [msg . sender ] -= amount ;
                balances [ receiver ] += amount ;
                emit Sent ( msg. sender , receiver , amount );
        }
}
```

La prima riga è una direttiva ***pragma*** che dice che il codice sorgente è scritto per Solidity versione 0.4.21, e il contratto può essere compilato dalla versione 0.4.21 in poi dell'EVM.
Il contratto ha le variabili di stato minter, che è un ***indirizzo*** Ethereum, balances, che è un ***mapping*** di indirizzi a interi senza segno, ovvero una struttura dati chiave-valore. Inoltre, viene dichiarato un evento, chiamato Sent, che prende l'indirizzo di mittente, destinatario e quantità da inviare. Ci sono poi le funzioni Coin, la quale è un costruttore. Nella versione 0.4.21 di Solidity non è ancora implementata la parola chiave ***constructor*** per i costruttori. La sintassi indicata nel contratto rappresenta una vulnerabilità perché il costruttore essendo pubblico può essere chiamato impropriamente, modificando la variabile minter. La variabile ***msg.sender*** rappresenta l'indirizzo di chi ha chiamato questo smart contract, il cui indirizzo viene associato alla variabile minter. La funzione mint controlla che chi ha chiamato il contratto sia diversa dal minter, e nel caso esce. Altrimenti, aumenta la variabile ***balances*** del receiver dell'ammontare richiesto. La vulnerabilità qui sta nel fatto che non vengono utilizzati contratti di libreria come ***SafeMath*** o Zeppelin per le operazioni matematiche, per cui potrebbero verificarsi overflow dovuti all'intervallo di rappresentazione del tipo della variabile. La terza funzione implementa il trasferimento di un saldo. Controlla che il saldo di chi ha chiamato il contratto è minore del saldo da inviare, se è così la funzione termina, altrimenti il saldo viene tolto al conto del mittente e aggiunto a quello del destinatario, dopodiché viene lanciato un evento che segna il ricevente, destinatario e saldo trasferito.