|  |
| --- |
| lab 1: solidity a smart contract language |
| ● Solidity lets you program on Ethereum, a blockchain-based virtual machine that allows the creation  and execution of smart contracts, without requiring centralized or trusted parties  ● Statically typed, contract programming language that has similarities to Javascript and C  ○ Like objects in OOP, each contract contains state variables, functions, and common data types  ○ Contract-specific features include modifier (guard) clauses, event notifiers for listeners, and custom global variables |
| language details:  event - publicize actions to external listeners  Constructor - can receive one or many variables here; only one allowed  msg provides details about the message that's sent to the contract  msg.sender is contract caller (address of contract creator)  balances[msg.sender], no this or self required with state variable  () : Fallback function - Called if other functions don't match call or sent ether without data |
| IDE: remix |
|  |
| lec 2: Ethereum mechanism |
| For Ethereum - a blockchain is a “cryptographically secure transactional singleton machine with shared-state.”  Cryptographically secure - Can’t create fake transactions, erase transactions  Transaction singleton machine - single instance of the machine for all the transactions  Shared-state - state stored on this machine is shared and open to everyone |
| Ethereum blockchain is essentially a transaction based state machine |
| Two Types of Accounts:  ○ Externally owned, which are controlled private keys and have no code associated with them.  ○ Contract accounts, which are controlled by their code and have code associated with them. |
| Account state content:  ● nonce: number of transactions sent from external account, number of contracts created if contract account  ● balance: The number of Wei owned by this address. 1e18 Wei per 1 Ether.  ● storageRoot: A hash of the root node of a Merkle Patricia tree. This tree encodes the hash of the storage contents of this account, and is empty by default.  ● codeHash: The hash of the EVM (Ethereum Virtual Machine — more on this later) code of contract account account. Hash(“”) for external accounts. |
| Any action that occurs on the Ethereum blockchain is always set in motion by transactions fired from externally controlled accounts |
| State of all accounts is the state of the Ethereum network |
| Ethereum network state is updated with every block  ○ A block takes the previous state and produces a new network state  ■ every node has to agree upon new network state |
| pros: space saving, more intuitive, comparable efficiency due to merklization |
| Not enough gas to execute the transaction?  ○ Transaction runs out of gas and therefore is considered invalid  ○ State changes are reversed, failing transaction recorded  ○ Since computation was already expended by the network, none of the gas is refunded |
|  |
| Gas is also used to pay for storage |
| Smart contracts are therefore best used for simple tasks  ○ Business logic or verifying signatures rather than machine learning or file storage  ○ Redundantly parallel, no asynchronous or performant parallel execution |
| Two types of transactions: Message calls and contract creations  ● nonce: number of transactions sent by sender  ● gasPrice: amount of Wei sender is willing to pay per unit of gas required to execute the transaction  ● gasLimit: max amount of gas the sender is willing to pay for executing this transaction, set before any computation is done  ● to: address of the recipient  ● value: the amount of Wei to be transferred from the sender to the recipient  ● v, r, s: used to generate the signature that identifies the sender of the transaction.  init (only exists for contract-creating transactions): An EVM code fragment that is used to initialize the new contract account  ● data (optional field that only exists for message calls): the input data (i.e. parameters) of the message call |
| Contracts that exist within the global scope of Ethereum can talk to other contracts using  messages (internal transactions) to other contracts  We can think of messages as being similar to transactions, except they are not generated by externally owned accounts, only by contracts |
| Messages do not contain a gasLimit |
| What if we run out of gas within a parent execution?  ○ Current and subsequent message executions will revert, however the parent execution need not revert |
| Contracts in Ethereum are like autonomous agents that live inside of Ethereum network  ● React to external world when "poked" by transactions (which call functions)  ● Have direct control over:  ○ internal ether balance  ○ internal contract state  ○ permanent storage |
| Ethereum Contracts generally serve four purposes:  ○ Store and maintain data  ■ representing something useful to users or other contracts  ■ ex: a token currency or organization’s membership.  ○ Manage contract or relationship between untrusting users  ■ ex: financial contracts, escrow, insurance.  ○ Provide functions to other contracts, serving as a software library.  ○ Complex Authentication  ■ ex: M-of-N multisignature access |
|  |
| lab 2: |
| Truffle, ganache |
| Contracts: contains our source code  Migrations: add scripts for staging deployment tasks  Test: where you test your contracts for vulnerabilities |
|  |
| lec 3: smart contract security |
| blockchain as a server, where server side code is completely exposed to the clients on the protocol (network nodes) and clients on DApps (users) |
| Attributes:  Smart contracts are immutable  ○ Once deployed, you cannot change their code  ○ Therefore you cannot fix the discovered bugs in the contract |
| Solutions:  Better testing methods for smart contracts  Formally verifiable smart contracts  Smart contract code audits  Easier smart contract languages with safety built in  Have very careful programmers write code on this platform |
| Pitfalls:   1. integer overflow/underflow   Mitigation: OpenZeppelin’s Safemath library   1. visibility   Keep your functions private or internal unless there is a need for outside interaction.   1. Fallback function   To receive ether for this function, we must mark it as payable   1. Delegatecall   parity hack   1. Call comparison   delegatecall, callcode, call   1. Reentrancy attack   solution 1: Reduce the sender’s balance before making the transfer of value  solution 2: Use mutexes to mitigate race conditions  solution 3: use require(msg.sender.transfer(\_value))   1. Selfdestruct   Fix: Never use a contract’s balance as a guard   1. DoS with unexpected revert 2. Short address attack |
| Everything on the EVM is deterministic |
| Best practice:   1. use send() over call.value()   the called contract is only given 2300 gas against reentrancy   1. require(condition) is meant to be used for input validation 2. assert(condition) should only be used for internal errors or check invalid state 3. lock pragmas to specific compiler version 4. all integer division rounds down to the nearest integer 5. remember on-chain data is public 6. beware of the possibility that some participants may drop offline and not return 7. test driven development |
|  |
| lec 4: testing and tokens |
| javascript syntax:   1. How to print? Use console.log(‘...’) 2. 'use strict';? Strict mode 3. Fallback function: function getInput (options, callback) {   allUserData.push (options);  callback (options);  }   1. let vs. var: var global access, let local scope 2. const variable cannot change |
| Mocha syntax:   1. describe, it:   ○ describe is used to group tests together by some criteria  ○ it is used to define a test case  2. before, beforeEach, after, afterEach  ○ These are hooks to run before/after first/each it or describe  ● contract(‘NameOfSuite’, function(accounts)) {...}  ○ Before each contract function is run, your contracts are redeployed to the running Ethereum client so the tests within it run with a clean contract state  ○ The contract function provides a list of accounts made available by your Ethereum client which you can use to write tests  ● artifacts.require("./contract.sol");  ○ Because Truffle has no way of detecting which contracts you'll need to interact with within your tests, you'll need to ask for those contracts explicitly  ● web3.eth.getBalance  ● .call(...)  ○ Used to specify that a method is explicitly NOT a transaction, e.g. a getter method  ○ Transactions will not execute if .call(...) is used  ● function(accounts)  ○ This is used to reference Ganache accounts. By default, accounts[0] is what is used for methods  ● contract.address  ○ Default way to get a contract’s address on the network  ● .valueOf()  ○ Used to retrieve number from a contract balance  ● {from: someOther.address, value: someAmount}  ○ from  ■ Usually executed from EOA’s for testing  ■ Only non-transaction calls work for contract addresses  ○ value  ■ Specify some amount of wei to send from an account’s balance  ■ Does not work if the account does not have enough value |
| CHAI assertions |
| Asynchronous programming:  Promise: then, async/await, |
| Tokens:  Augur (REP) is an example of an application built on top of protocols:  ● Decentralized Oracle Protocol  ○ REP provides a financial incentive for a network of nodes to arrive at a consensus around real-world happenings  ● Exchange Protocol  ● Protocol tokens provide the financial incentives needed to drive a cryptoeconomic  protocol which may or may not be implemented within an Ethereum smart contract  ● DApps act as access points into protocols but have no cryptoeconomic backbone; App  coins align dApp developer and investor incentives  ● Redundancy introduces unneeded costs for end users and causes a splintering |
| The ERC20 standard: rules for Ethereum tokens to follow |
|  |
| lab 4: launching an ICO |
| Etherscan  ERC223/827:  Created: 2017-03-05, aimed to fix major flaws.  ● Impossibility of handling incoming tx in receiver contract  ● Tokens could be sent to contract that is not designed to work with tokens and potentially could be lost  ● Token-transactions should match Ethereum ideology of uniformity. When a user needs to transfer their funds, they must always perform transfer. Uniform across sending to an externally owned account  ERC721  the “non-fungible” token model |
| MINIME tokens:  clone other tokens: Generating a voting token that is burned when you vote |
| Extra tools:  Openzeppelin (include functionality like safe math, tokens, …) |
|  |
| lec 5: web3: interacting with Ethereum |
| web3.js is the entrance to the Ethereum blockchain from the client side of a Ðapp |
| The Ethereum ABI expose contract methods  can be generated by Remix |
| ● It is the official DApp API that is run on all of the Ethereum nodes  ○ Interfaces with Ethereum nodes from the network using JSON-RPC calls  ● Main Capabilities:  ○ Interact with contract functions  ○ Deploy contracts  ○ Send raw transactions to contracts with extra data  ○ Query the blockchain for data  ■ Includes logged events by any contract along with block data  ● Can have our back-end on chain, and our interface off chain  ● Use GitHub pages to host my apps with server level functionality? |
| infura |
| Testnets:  ● Identical to the mainnet functionality  ○ Copy of the protocol with controlled risk  ○ Sometimes uses different consensus  ● Minor difference in network/client parameters  ○ Different genesis block  ○ Different network ID  ○ Lower block difficulty |
| Geth vs Parity |
| public networks: options:  ● 0: Olympic, Ethereum public pre-release testnet  ● 1: Frontier, Homestead, Metropolis, the Ethereum public main network  ● 1: Classic, the (un)forked public Ethereum Classic main network, chain ID 61  ● 1: Expanse, an alternative Ethereum implementation, chain ID 2  ● 2: Morden, the public Ethereum testnet, now Ethereum Classic testnet  ● 3: Ropsten, the public cross-client Ethereum testnet  ● 4: Rinkeby, the public Geth Ethereum testnet  ● 42: Kovan, the public Parity Ethereum testnet  ● 7762959: Musicoin, the music blockchain |
| Proof of Work cannot work securely in a network with no monetary value! |
| Rinkeby: Newer, official Ethereum testnet  ○ Uses Proof of Authority  ■ Rely on trusted validators to ensure that valid transactions are  added to blocks, processed and executed by the EVM faithfully  ■ 4 second block time  ○ Requires geth (or any other client that supports clique consensus |
| Dapp architecture:  client-blockchain in serverless apps: entire flow happens between the client and blockchain    If you don’t want to host it yourself, you can leverage Infura, a service that offers public nodes at no cost  With Mist and Metamask, this is easy, both provide a way to manage the user accounts and request the user to approve a transaction when the application is asking for ETH  ● If you don’t require your user to install Metamask or Mist to use your app, you’ll need to direct them to manually send the transactions from whichever wallet they work with  ● Most applications will implement this by asking the user to send a certain amount of ETH to an address, optionally including a scannable QR code or copy-to-clipboard button |
| server to blockchain model  ● Suppose we add a server to the mix and leave the client aside  ○ The server in this case can be an application server, scripts or batch processes  ● Option 1: Local Node  ● Option 2: Offline signing and public nodes |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |
|  |