Practical No : 3

Title : Write a smart contract on test network, for Bank account of a customer for following operations:
Deposit money
Withdraw money
Show balance.

Objective : Student should be able to learn new technology such as metamask. Its application & implementations.

Prerequisite : 1. Basic knowledge of cryptocurrency
2. Basic knowledge of distributed computing concept
3. working of blockchain.

Theory :
The contract will allow deposits from any account, & can be trusted to allow withdrawals only by accounts that have sufficient funds to cover the requested withdrawal.

• This post assumes that you are comfortable with the ether handling concepts introduced in our post. writting a contract that handles Ether.

That post demonstrated how to restrict ether withdrawals to an owners account. It did this by persistently storing the owner account address & then comparing it to the msg. sender value for any withdrawal attempt.

- I am going to generalize this contract to keep track of ether deposits based on the account address of the depositor, & then only allow that same account to make withdrawals of that Ether. To do this we need a keep track of account balances
- for each depositing account - a mapping from accounts to balances.

- which will make this book keeping job quite simple. Here's the code to accept deposits & track account balances.

```
Program solidity ^0.4.19
contract Bank {
mapping (address => unit 256) public balance of;
function deposit (unit 256 amount ) public payable {
require(msg.value == amount ) ;
balanceof [msg.sender] += amount ;
}
}
```

It's important to note that balanceOf keeps track of the ether balances assigned to each account, but it does not actually move any ether anywhere, The bank contract's ether balance is the sum of all the balances of all accounts - only balanceOf tracks how much of that is assigned to each account.

- Note also that this contract doesn't need a constructor there is no persistent state to initialize other than the balanceOf mapping, which already provides default values of 0.

- Given the balanceOf mapping from account addresses to ether amounts, the remaining code for a fully functional bank contract is pretty small I'll simply add a withdrawal function:

- In the withdraw() function above, it is very important to adjust balanceOf[msg.sender] before transferring ether to avoid an exploitable vulnerability. The reason is specific to smart contract & the fact that a transfer to smart contract executes code in that smart contract.

- Now, suppose that the code in withdraw() did not adjust balanceOf before making the transfer & suppose that msg.sender was a malicious smart contract

Upon receiving the transfer - handled by msg.sender's fallback funcⁿ that malicious contract could initiate another withdrawal from the banking contract. when the banking contract handles this second withdrawal request, it would have already transfered ether for the original withdrawal, but it would not updated balance, so it would allow this second withdrawal.

- This vulenerability is called a reentracy bug because it happens when a smart contract invoke code in a different smart contract that then calls back into the original thereby reentering the exploitable contract.

- To avoid ~~the~~ this sort of reentracy bug, follow the 'check - effects - Interactions pattern' as described in the solidity documentation. The withdraw() funcⁿ above is an example of implementing this pattern.

Practical No: 4

* Title: Write a program in solidity to create student data. Use the following constructs:
  • structures • Arrays • fallback
  Deploy this as smart contract on Ethereum & observer the transaction fee & Gas values.

* objective: Students should be able to learn about solidity. Its datatypes & implementation.

* Prerequisite: 1. Basic programming logic
  2. Basic knowledge of solidity.

* Theory

1. Solidity - Arrays:
Arrays are data structures that store the fixed collection of elements of the same data types in which each & every element has a specific location called index. Instead of creating numerous individual variables of same type, we just declare one array of the required size & store the elements in array & can be accessed using the index.

creating an Array: Synax:
<data type> <array name> [size] = <initialization >

- Fixed size Array : The size of array should be predefined The total number of elements should not exceed the size of the array.

- Dynamic Array : The size of array is not predefined when it is declared. As the element are added the size of array changes & at the runtime, the size of the array will be determined.

Array Operations :

1) Accessing Array elements : The elements of the array are accessed by using the index. If you want to access ith element then ~~h~~ you have to access (i-1) the index

2) PUSH : push is used when a new element is to be added in a dynamic array. The new element is always added at the last position of the array.

3) Pop : Pop is used when the last element of the array is to be removed in any dynamic array.

Solidity - Structures :

- Structs in solidity allows you to create more complicated data types that have multiple properties. You can define your own type by creating a struct.

Syntax : struct < structure-name > {
<data type > variable-1;
<data type > variable-2; }

* Solidity fallback :
The solidity fallback func^n is executed if none of the other func^n match the func^n identifier or no data was provided with the func^n call.

* Properties of fallback func^n:
- Has no name or arguments.
- if it is not marked payable, the contract will throw an receives plain ether without data.
- can not return anything
- can be defined once per contract
- It is also executed if the caller meant to call a func^n that is not available.
- It is mandatory to mark it external
- It is limited to 2300 gas when called by another func^n. It is so for as to make this func^n call as cheap as possible.

* Conclusion :
In this way we have created array, structure & used fallback func^n in solidity.