Merkle Airdrop Contract Deployment Instructions

Introduction

This instruction will walk you through the process of deploying Merkle Airdrop contract using Remix IDE. See attachments for contract source code.

Overview

The process consists of the following steps:

1. Set up dependencies and tools
2. Paste contract code to Remix IDE
3. Compile Contract Code
4. Calculate Merkle Root Hash
5. Deposit Funds to Airdrop Contract
6. Claim Airdrop Process
7. RecoverERC20 Function

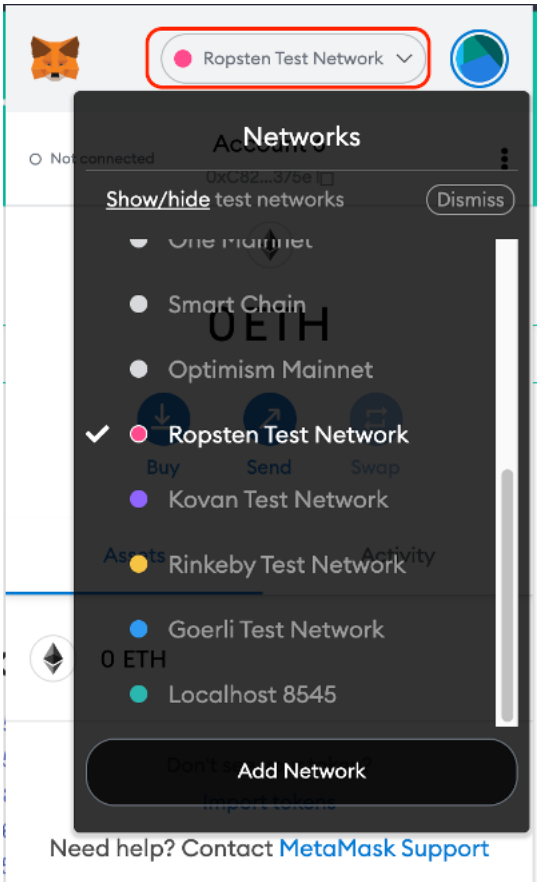
Dependencies and Tools

To follow each of these steps, you will need certain tools:

1. Installed Metamask and available Ether (see instructions here)
2. Remix IDE (https://remix.ethereum.org/)

1. Set-up Dependencies and Tools

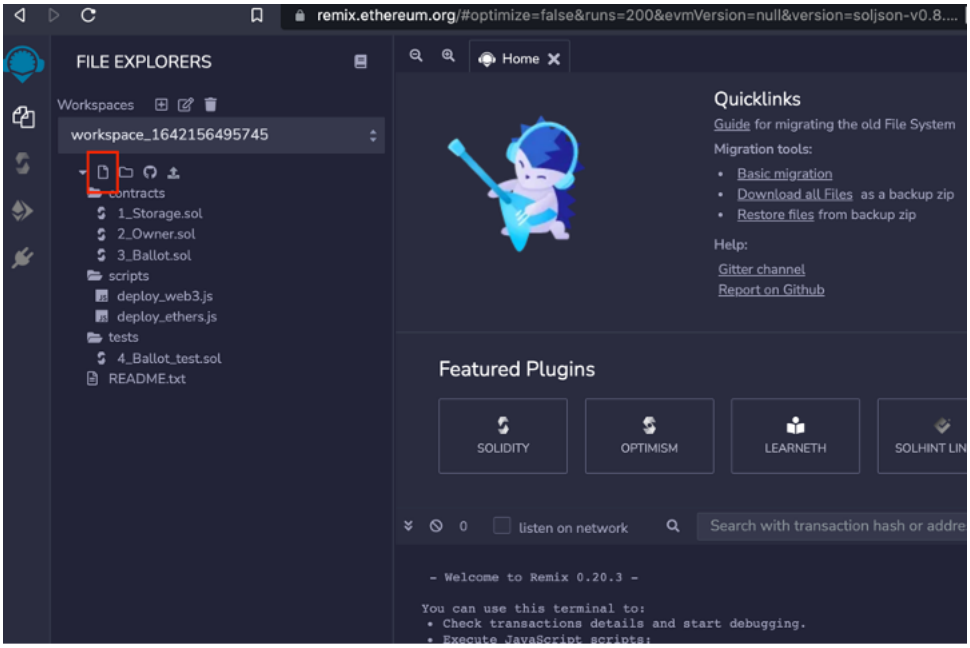
Make sure you have installed Metamask extension in your browser and have ETH available to pay for deployment cost. For this example, we will be using Ropsten testnet so you will need to change Metamask network by clicking on Netwok selector at top of Metamask interface.



You can obtain free test Ether here: https://faucet.ropsten.be/
Next you need to navigate to Remix IDE where we will deploy smart contract (https://remix.ethereum.org/). Get a copy of smart contract which we'll use for deployment (see attachment). Lastly, this instruction uses Remix text editor, and Web3.Utils Javascript library for Merkle proof algorithm scripting, but you may use other tools you feel more comfortable with.

2. Paste Contract Code to Remix IDE

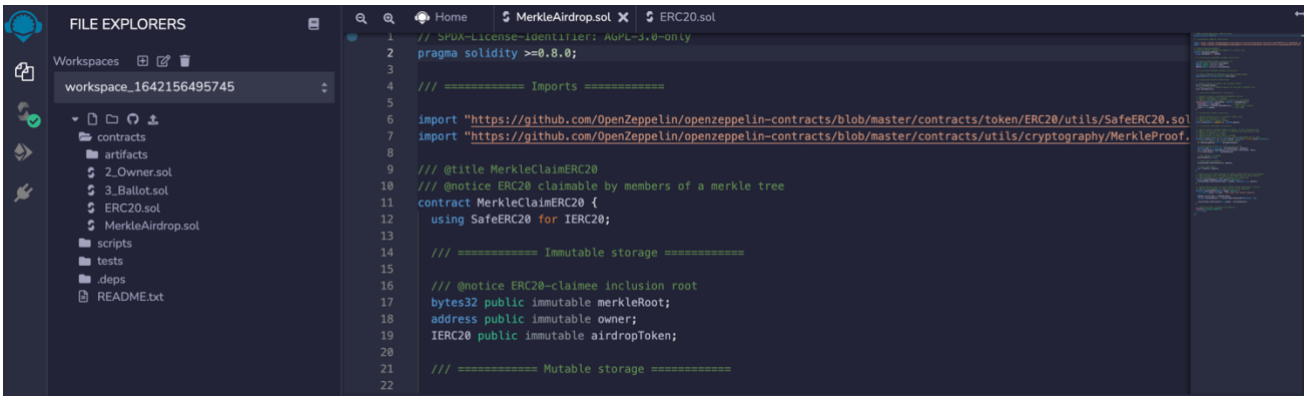
Navigate to Remix IDE and create and new file under your workspace. You will need to name it, in this example, I will name contract "MerkleAirdrop.sol"



Next, paste the Airdrop smart contract source code into Remix. Ensure the import lines point to OpenZeppelin contract library:

```
/// ============ Imports ============

import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/token/ERC20/utils/SafeERC20.sol"; // OZ: IERC20
import "https://github.com/OpenZeppelin/openzeppelin-
contracts/blob/master/contracts/utils/cryptography/MerkleProof.sol"; // OZ: MerkleProof
```

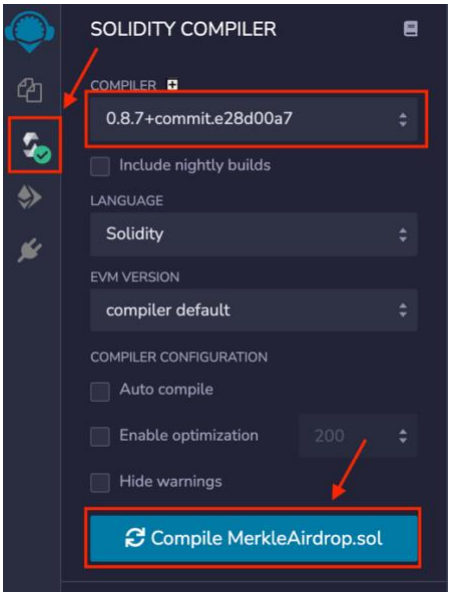
Your contract should look like this:



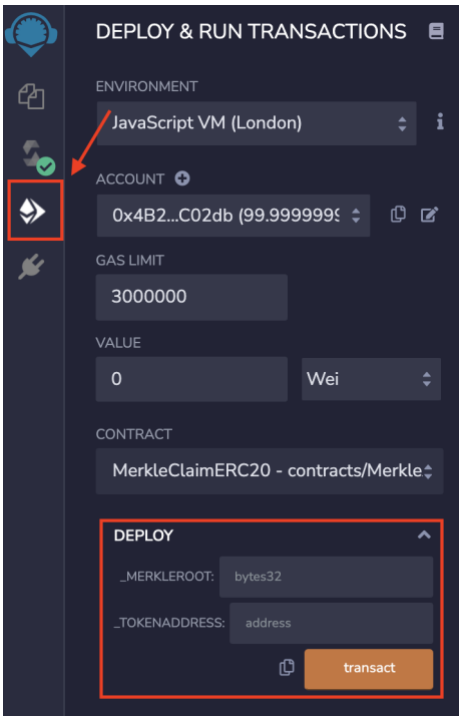
For the purpose of this exercise, we will need a basic ERC20 token to be airdropped. Hence, we will create a simple token with a supply of 1,000 tokens.

3. Compile Contract Code

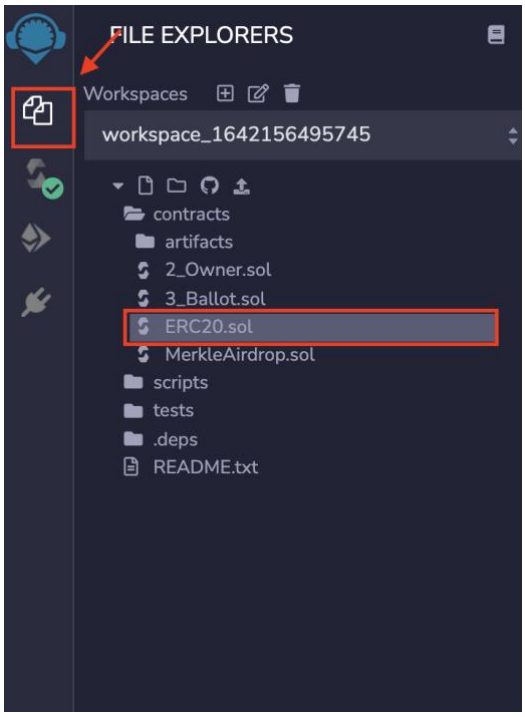
We are almost ready to deploy contract to the testnet. First, click on Compiler tab on the left-hand menu. Here we select compiler 0.8.7+commit.e28d00a7 and then click blue button "Compile MerkleAirdrop.sol".



Lastly, we navigate to Deploy & Run Transaction tab on left-hand menu. We will have an orange button "Deploy". Click on the down arrow on the right to bring up input field for all parameters required for deployment; bytes32 _MERKLEROOT and address _TOKENADDRESS.



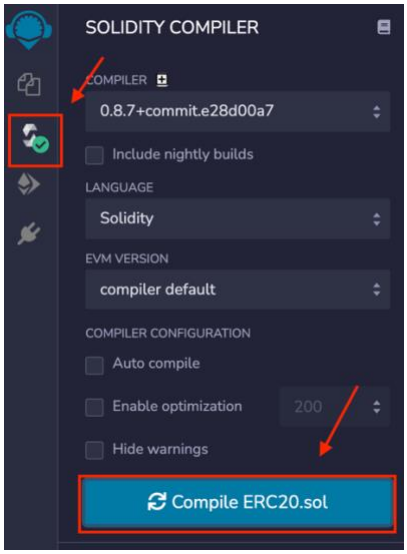
Since we will be using a sample ERC20 token for this example, let's go ahead and create it. First copy and paste the token source code in new file as done in step 2



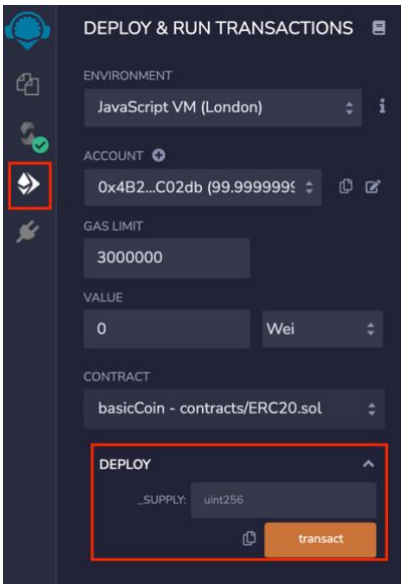
Source code:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/ERC20.sol";

contract basicCoin is ERC20 {

    constructor(uint256 _supply) ERC20("myCoin", "MyC") {
        _mint(msg.sender, _supply);
    }
}
```

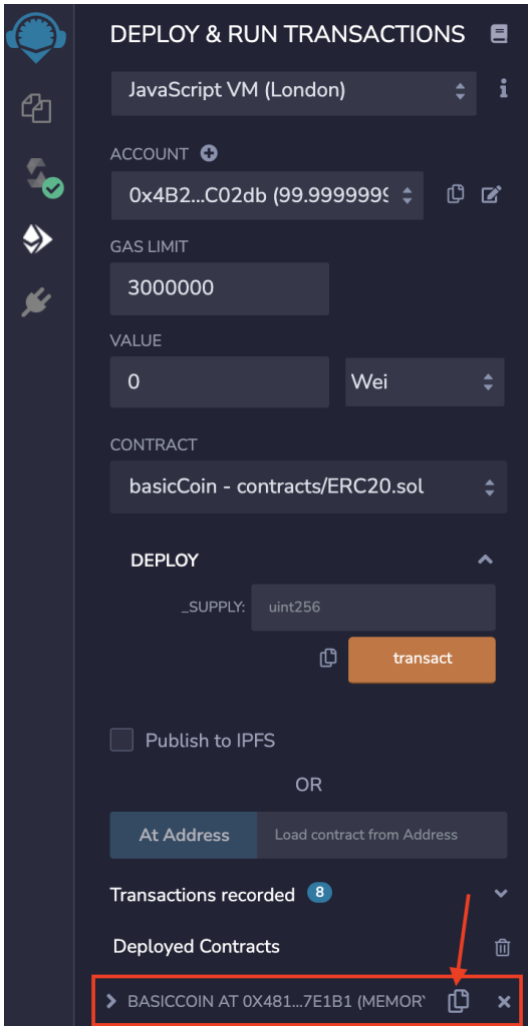
We compile the source code



Finally, we deploy it with _SUPPLY = 1000



After deployment, you will see contract at the bottom, and we can copy the contract address. In my case I received contract address
0x4815A8Ba613a3eB21A920739dE4cA7C439c7e1b1



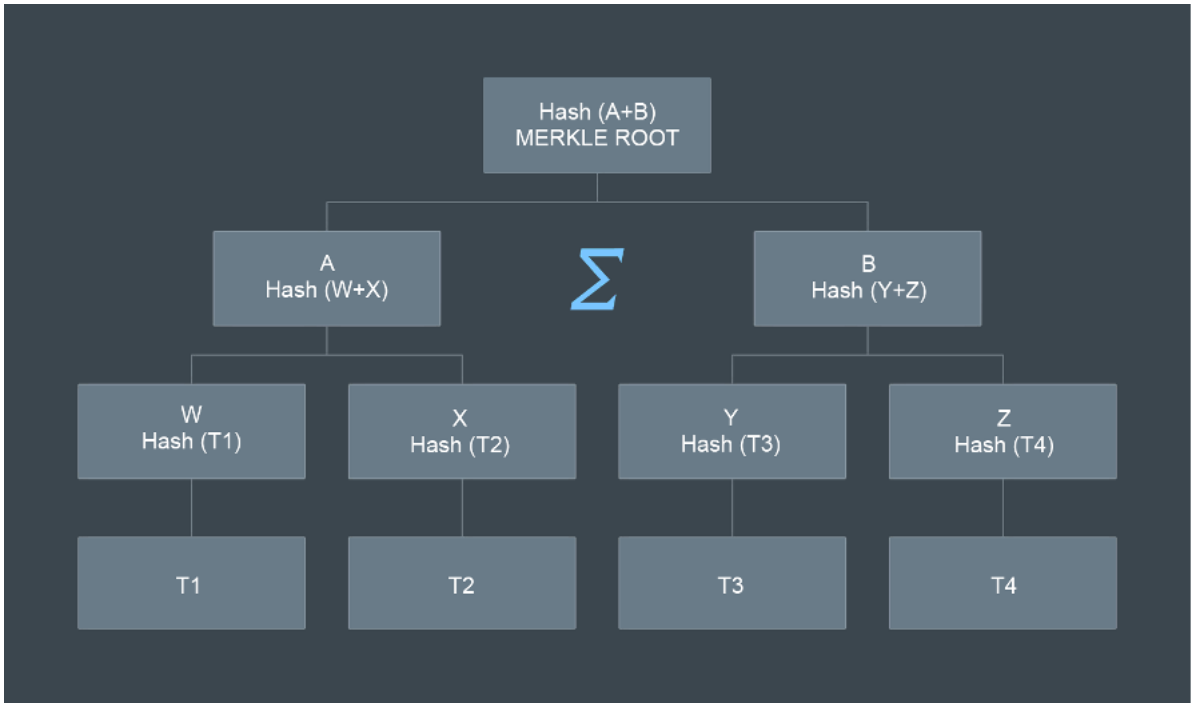
We will use this contract address for the deployment of MerkleAirdrop contract (_TOKENADDRESS). Now we just need to compute the Merkle root hash.

---

4. Calculate Merkle Root Hash

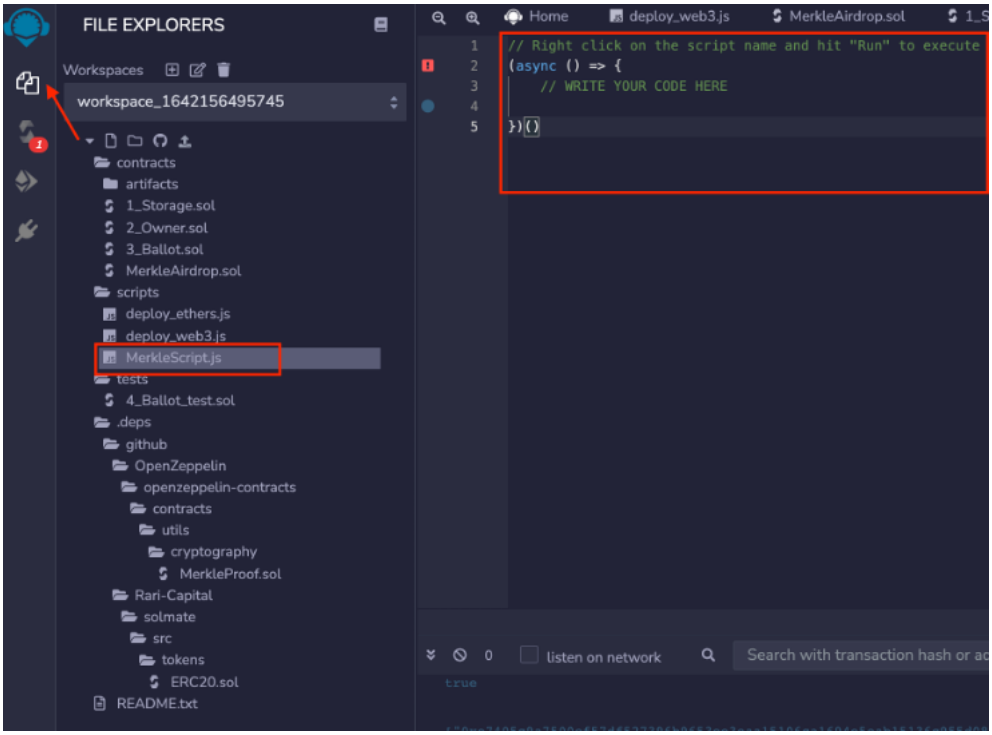
In the previous section we created a token for the airdrop but in your case, you may already have your ERC20 token. The token needs to be created prior to deploying the airdrop contract. The more challenging parameter to determine is the root hash which will determine who is able to claim airdrop tokens. First, we need a list of balances consisting of an address, amount pairs describing which accounts are eligible to claim airdrop and how much they can claim. For an introduction to Merkle Trees and Merkle Root you can read article here.

In short, we will hash each balance pair using keccak256 hashing function, then we will recursively hash pairs together until we have only one hash which we will call the Root hash. The process looks like the below.



To hash data using keccak256 we use Web3.utils.soliditySha3() function which mimics the same algorithm used by Solidity.

You can write your script on Remix by creating new file under folder "scripts". In my example, I have added a new file and called it MerkleScript.js. Remix IDE already import Web3 package so we don't need to add anything else to make use of Web3.utils.soliditySha3() function. We can input our Javascript code and right click on the script file and hit "Run" to execute the script.



For this example, I will use the following balances for the airdrop.

```
const AirdropRecipients = [
    {
        "to": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4",
        "amount": "500"
    },
    {
        "to": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2",
        "amount": "200"
    },
    {
        "to": "0x4B20993Bc481177ec7E8f571ceCaE8A9e22C02db",
        "amount": "300"
    },
    {
        "to": "0x0000000000000000000000000000000000000000",
        "amount": "0"
    }
];
```

Important to note is that you need to have an even number of objects for Merkle Tree calculation. Since we need to hash each element is pairs of two, we need to ensure the number of elements is never odd, even if it means duplicating objects or adding dummy hashes. In my example I had 3 airdrops (taken from Remix top 3 test accounts) and I added a dummy 0 address airdrop to make it even. This is applicable at every height of the Merkle Tree.

Now we hash each of the balances into keccak256 hashes:

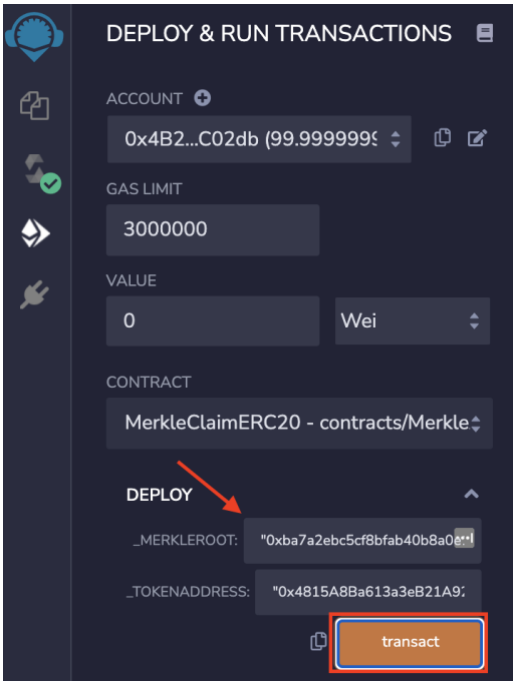
```
function createLeaves(airdrop) {
    var leaves = [];
    for(i = 0; i < airdrop.length; i++){
        let hash = Web3.utils.soliditySha3(airdrop[i]['to'], airdrop[i]['amount']);
        leaves.push(hash);
    }
    return leaves.sort();
}

console.log(createLeaves(AirdropRecipients));
```

```
// OUTPUT:
//[
// "0x05034220bcab641e4bd65ba93e544f366801b00fa92e80d9c801bdb920cf702b"
// "0x3101a918c0421f07f9f7ef6a8ef2d362afcbc05b9ffaf14d0d9545627e9a4608"
// "0x9da137fb3ca535f5dfcc27f9f4cf8ac9f375dca42354212f8545244ee36eabea"
// "0xa86d54e9aab41ae5e520ff0062ff1b4cbd0b2192bb01080a058bb170d84e6457"
//]
```

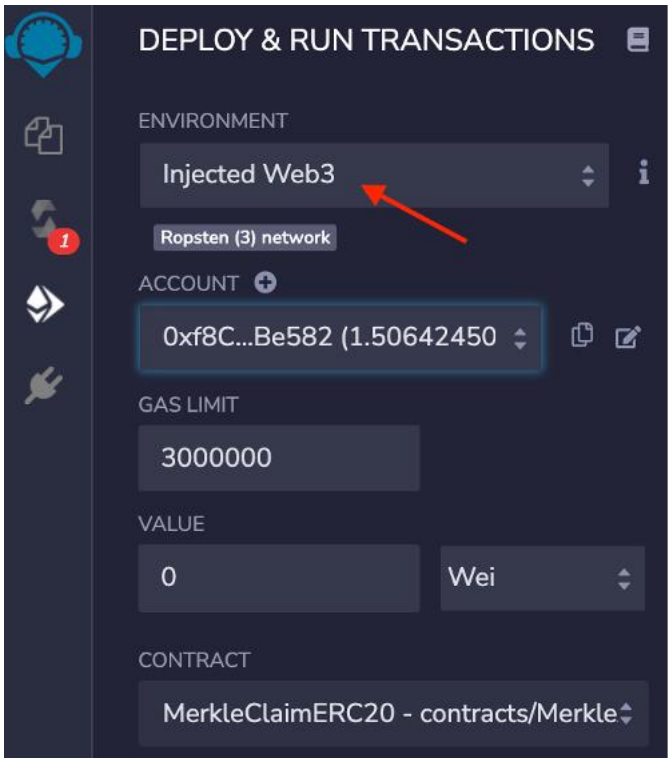
It is also important to note that MerkleProof.sol implementation by OpenZeppelin requires that all our hashes are sorted in ascending order. Hence, we sort all our hash using leaves.sort(). This creates our set of end points or "leaves". Then, we recursively hash our leaves in sibling pairs [hash(index(0), index(1)) ; hash(index (2), index (3))] and so on.

```
function calculateMerkleRoot(_leaves){
    let nodes = [];
    for(i = 0; i<=_leaves.length / 2; i+=2){
        let hash = Web3.utils.soliditySha3(_leaves[i], _leaves[i+1]);
        nodes.push(hash);
    }

    nodes = nodes.sort();
    if(nodes.length > 1){
        return calculateMerkleRoot(nodes);
    } else {
        return nodes;
    }
}

let leaves = createLeaves(AirdropRecipients);
console.log(calculateMerkleRoot(leaves));
// OUTPUT:
//[
// "0xba7a2ebc5cf8bfab40b8a0e1ad414aca3b29aa577f467c2facce598143f173e9"
//]
```

We can see our root hash is "0xba7a2ebc5cf8bfab40b8a0e1ad414aca3b29aa577f467c2facce598143f173e9" and this is what we need to input as our last parameter in the "Deploy" function. So, let's do that.



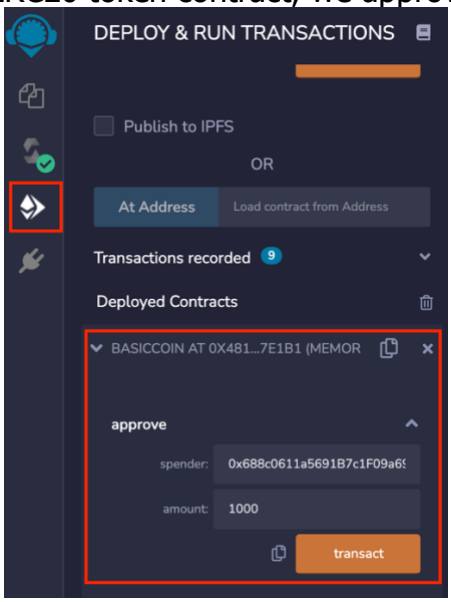
Fill in the required information and click "transact". This will deploy the contract to Remix Test environment. To deploy to Ropsten public testnet then in Deploy & Run Transactions tab, under "Environment" select Injected Web3. This will prompt Metamask to connect to the site. Click accept. Now, if you click "transact" once again, this will prompt a Metamask transaction to deploy the contract on blockchain. You will need Ropsten Ether to pay the transaction cost. Once the transaction is submitted and confirmed, your smart contract will be live on the testnet blockchain. To deploy to mainnet you can follow the same steps but changing Metamask network back to "Ethereum Mainnet".



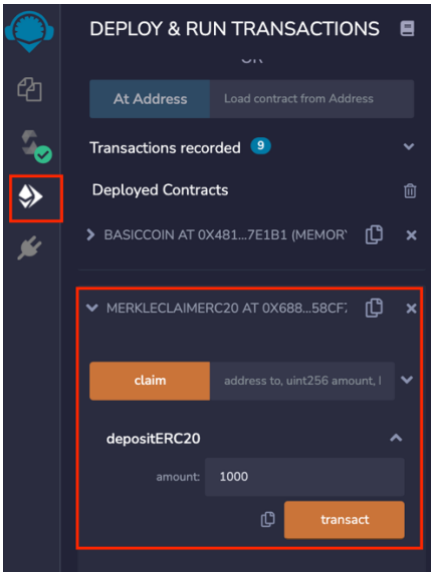
5. Deposit Funds to Contract

Now that the airdrop contract is live on blockchain, we need to provide it with tokens so that users are able to claim them. The airdrop contract doesn't know how many tokens it will need to hold to pay out all valid claims, so it is up to the contract deployer to make sure enough tokens are sent to the contract. In our case, we know we will need 1,000 tokens to fulfil all valid claims. We can fund our contract in two ways. First, we can directly send our tokens to the contract address, but this can be prone to error since we may send the incorrect tokens. A second way is to call the contract "depositERC20" function which will let the contract handle the transfer. For this, we need to first approve the airdrop contract to take our tokens.

From previous step, we found our airdrop contract address is 0x688c0611a5691B7c1F09a694bf4ADfb456a58Cf7 and we know we need to deposit 1,000 tokens. Therefore, using the ERC20 token contract, we approve the following:



Then we can go to airdrop contract and call "depositERC20" function with amount 1,000

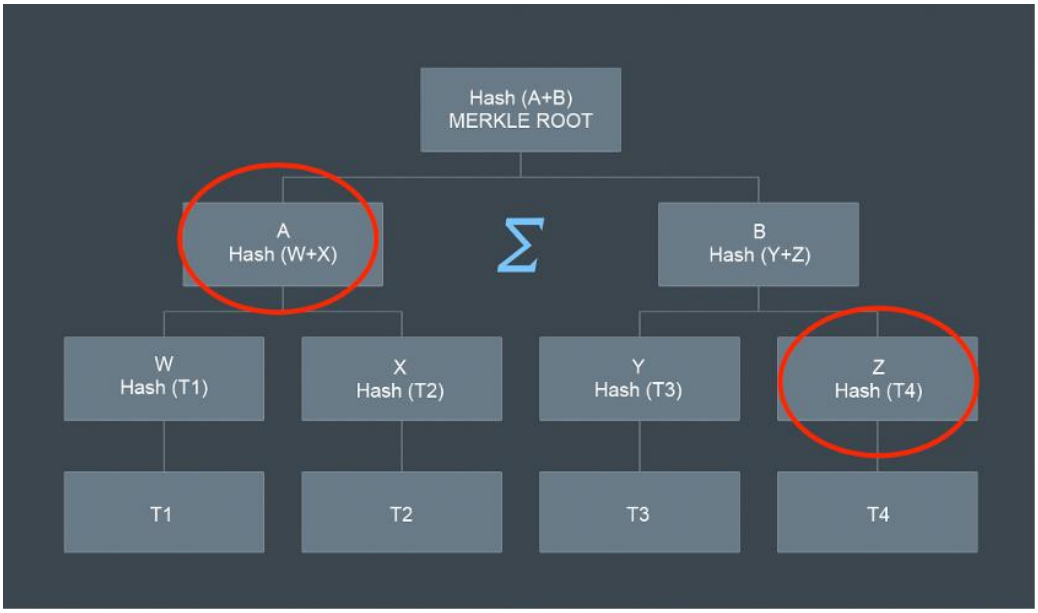


Now our airdrop contract is fully funded and participants will be able to claim the deposited tokens using the claiming process outlined below.

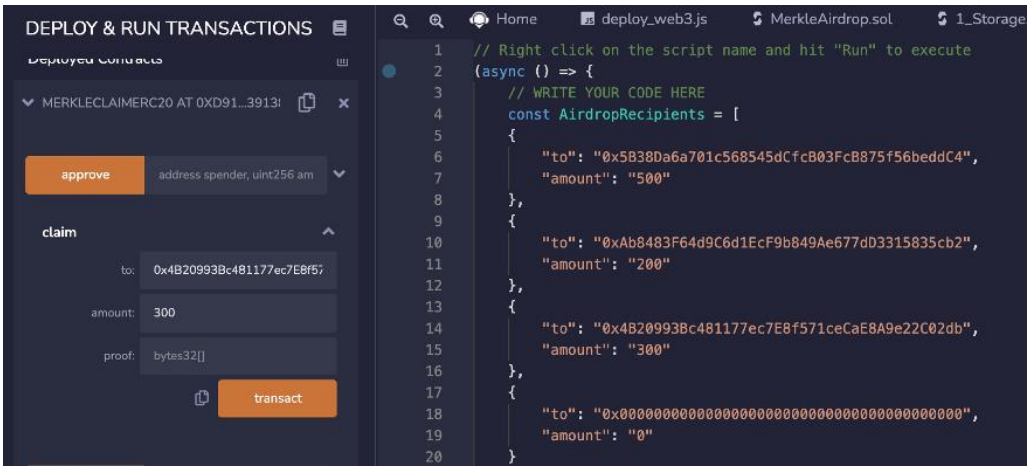
---

6. Claim Airdrop Process

Now that the contract is live on blockchain, users can interact with the contract and claim their share of their airdrop. However, the process requires some support from the airdrop sponsor. Whenever a user wants to claim their allocation of the airdrop, they need to provide "**proof**" that their requested balance was part of the balances used to calculate the Root hash obtained in Section 4.
The simplest way would be to provide the Airdrop contract a list of all valid balances along with our requested balance and have the smart contract re-calculate the root hash so that it can prove to itself that our airdrop balance was among the valid balances submitted by the airdrop sponsor. However, this is too computationally intensive and far too costly.
Luckily there's a shortcut to this process. If we refer to the computation of Merkle Root Hash, there's an easier to re-calculate the root hash. If we take our balance, let's say "T3", then if we can provide hash Z, we can compute hash B. Then if we also provide hash A, then together with hash B we can compute the Root hash. So, we only need two pieces of information instead of four. The benefit is that this process scales well for large number of airdrop balances, maintaining the computational cost reasonable for the smart contract by shifting the expensive computations off-chain (i.e. our front-end app).



Let's go through the process of claiming airdrop for account 3 from our list of accounts. After deploying your contract (either on testnet or mainnnet), you can interact with it via Remix on the same "Deploy & Run Transactions" tab. Here we find our function "claim" requires three parameters.

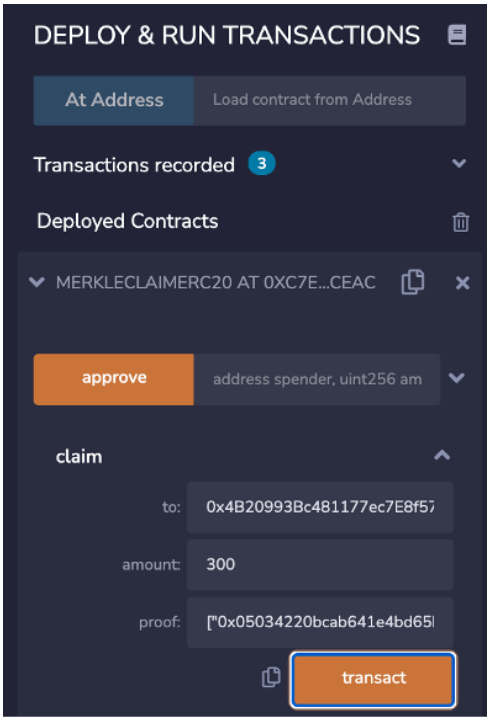


The first two parameters are easily obtained from our initial airdrop balances. The "proof" will need to be calculated by re-calculating the root hash and recording the hash for the "siblings" of transaction 3 at each height of the Merkle Tree.

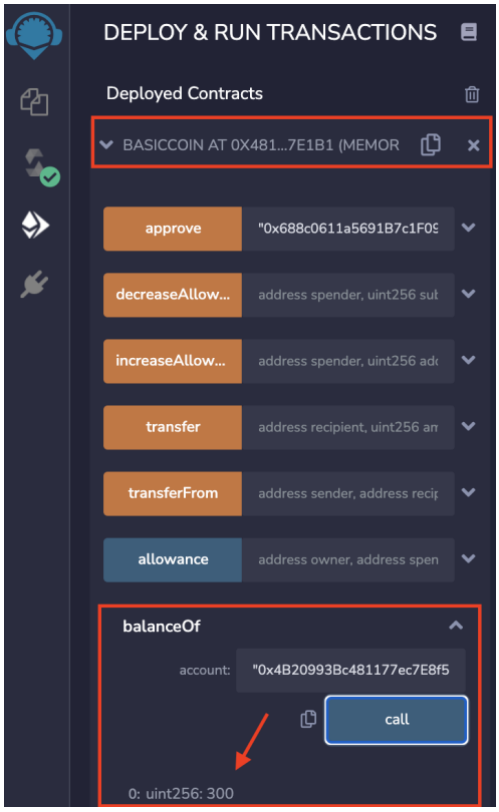
```
function getLeafAtIndex(index, airdrop) {
    if(index >= airdrop.length) return 0; // invalid index
    return Web3.utils.soliditySha3(airdrop[index]['to'], airdrop[index]['amount']);
}

function reduceMerkle(_leaves) {
    let nodes = [];
```

```
    for(i = 0; i<=_leaves.length / 2; i+=2){
        let hash = Web3.utils.soliditySha3(_leaves[i], _leaves[i+1]);
        nodes.push(hash);
    }
    return nodes.sort();
}

function calculateProof(leaf, leaves){
    let proof = [];
    let sortedIndex = leaves.indexOf(leaf);
    let sibling, hash;

    while(leaves.length >= 2){
        if(sortedIndex % 2 == 0){
            sibling = leaves[sortedIndex+1];
            hash = Web3.utils.soliditySha3(leaf, sibling);
        } else {
            sibling = leaves[sortedIndex-1];
            hash = Web3.utils.soliditySha3(sibling, leaf);
        }
        proof.push(sibling);
        leaves = reduceMerkle(leaves);
        sortedIndex = leaves.indexOf(hash);
    }
    return proof;
}

let leaves = createLeaves(AirdropRecipients);
let leaf = getLeafAtIndex(2, AirdropRecipients);
console.log(calculateProof(leaf, leaves));
// OUTPUT:
//[
// "0x05034220bcab641e4bd65ba93e544f366801b00fa92e80d9c801bdb920cf702b"
// "0x854590a373ce45f23569409355af366f4c4d19bfd8122e0762a7ff49e73f1502"
//]
```

Here we see the proof consist of two hashes, as expected. Let's try to input them in "claim" transaction.



After our transaction has been confirmed, we should have received 300 MyC tokens. Let's go back to the token contract and check our balance.



Yes, we can see our account balance correctly reflect 300 tokens!

7. Recover ERC20 Function

The Merkle Airdrop includes a ERC20 recovery function to allow contract deployer to remove any ERC20 tokens sent to Airdrop contract address. Airdrop contract is designed to airdrop other existing ERC20 tokens that require to be deposited into the contract. If airdrop participants do not claim their tokens, then the contract deployer can withdraw any remaining tokens from the contract so that they are not lost.

The function is defined as follow:

```
/// @notice Allows owner to recover ERC20 tokens deposited to contract
 /// @param token address of token contract to be recovered
 function recoverERC20(address token) external {
  require(msg.sender == owner, "Only owner can recover tokens");

  IERC20 recoverToken = IERC20(token);
  uint256 recoverBalance  = recoverToken.balanceOf(address(this));

  recoverToken.safeTransfer(msg.sender, recoverBalance);
 }
```

Fallback function to prevent accidental deposits of native ETH coins.

```
/// @notice Prevent accidental ETH deposits
receive() external payable {
    revert();
}
```

Use:

Firstly, we maintain the deployer's address at the time of deployment to ensure only the deployer address can call the recoverERC20 function. Then, in the recoverERC20 we check that the function caller is indeed the deployer address, otherwise transaction fails. To recover the ERC20 token we need to pass the token address as a parameter.

For example, let's consider the case that 100 Wrapped Ether (address 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2) has been deposited to the airdrop contract. In this case, the deployer will pass the address as the parameter, the contract will

create an instance of the WETH contract, check the airdrop contract's remaining balance and will send the tokens to the deployer's address. After this executes, the airdrop contract balance will go from 100 WETH to 0, and the deployer's address will reflect +100 WETH tokens.

Lastly, we add a fallback function to prevent deposits of native ETH coins. The airdrop contract and recoverERC20 are both designed to work with ERC20 tokens. However, since ETH coins are not ERC20 tokens, they cannot be recovered through recoverERC20 function hence we prevent any ETH deposits in the first place to protect users' funds. If deployer wishes to airdrop ETH tokens, then the Wrapped Ether (ETH ERC20 version) can be used instead.