Sri Lanka Institute of Information Technology

IT21299902 (ZAKEY M.S.M.A)

Smart Contract Competition

ETHERNAUT CTF

Web security – IE2062

B.Sc. (Hons) in Information Technology Specialization in cyber security.

Declaration:

- I hereby certify that no part of this assignment has been copied from any other work or any other source. No part of this assignment has been written/produced for me by another person.

- I hold a copy of this assignment that I can produce if the original is lost or damaged.

Case Study	Smart contract ctf.
Date Of completion	25/02/2023

Project Details:

Introduction.

Smart contracts are self-executing contracts that leverage blockchain technology to automate business processes and enable decentralized applications (dApps). They are written in programming languages like Solidity and deployed on blockchain platforms such as Ethereum. To use smart contracts, developers need to understand the coding concepts, syntax, and interactions with the blockchain network.

Benefits of smart contracts include transparency, efficiency, cost-effectiveness, and increased security. However, securing smart contracts is paramount due to potential vulnerabilities. Best practices include thorough testing, code audits, and adherence to coding standards. Smart contract competitions, such as Ethernaute CTF, are popular in the blockchain community, challenging participants to identify and exploit vulnerabilities in smart contracts to win prizes.

Ethernaute CTF is a well-known smart contract competition, and the author of the report plans to participate in it, showcasing the significance of such competitions in the field of blockchain technology. By understanding the fundamentals of smart contracts, their usage, benefits, security considerations, and participation in competitions like Ethernaute CTF, we can better appreciate the potential of blockchain technology and its impact on various industries.

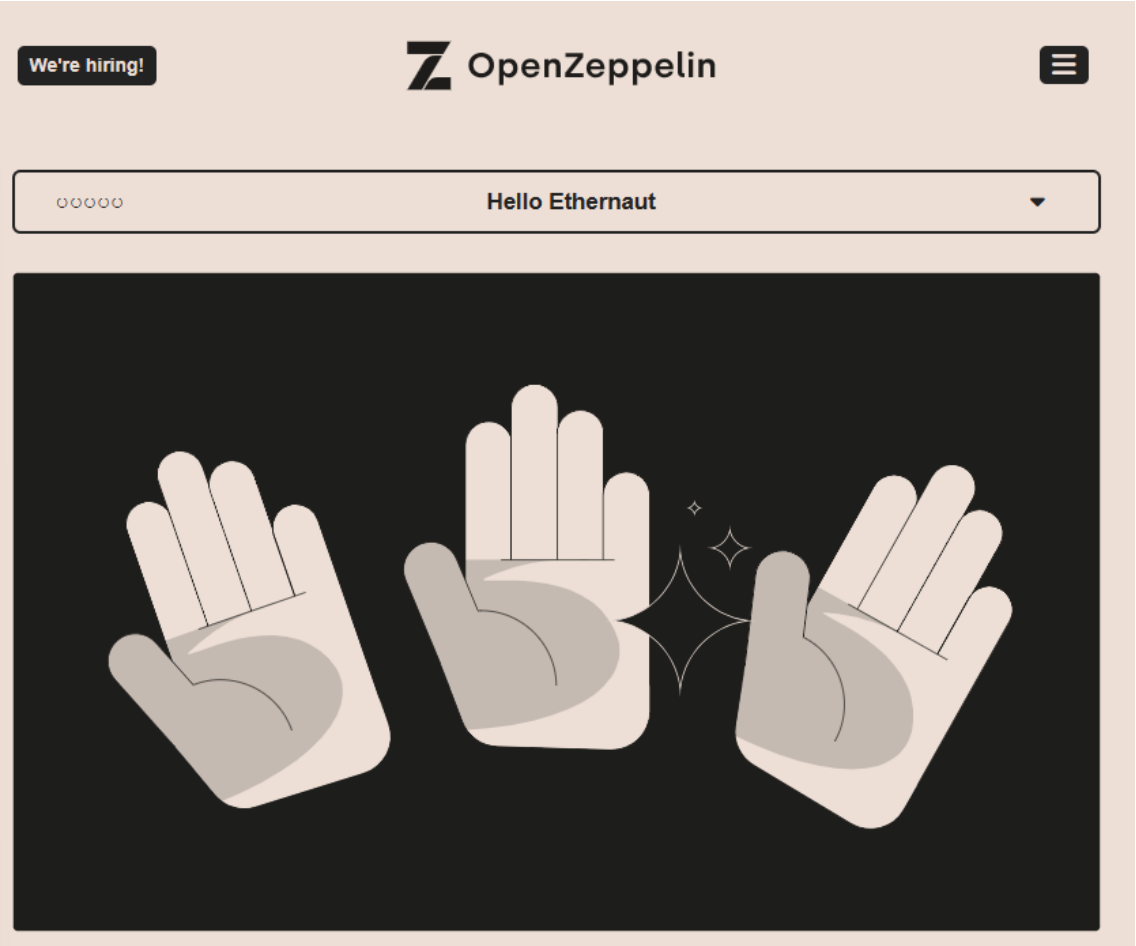
Introduction Ethernaute CTF.

Ethernaute CTF is a popular smart contract competition that challenges participants to test their skills in identifying and exploiting vulnerabilities in smart contracts. The competition is typically structured into multiple levels, each with increasing difficulty.

Participants are required to analyze and interact with smart contracts written in Solidity, identify potential security flaws, and craft exploits to breach the contract's defenses. The competition may involve tasks such as reverse engineering, code analysis, and exploit development to gain unauthorized access to contract functions or manipulate contract state.

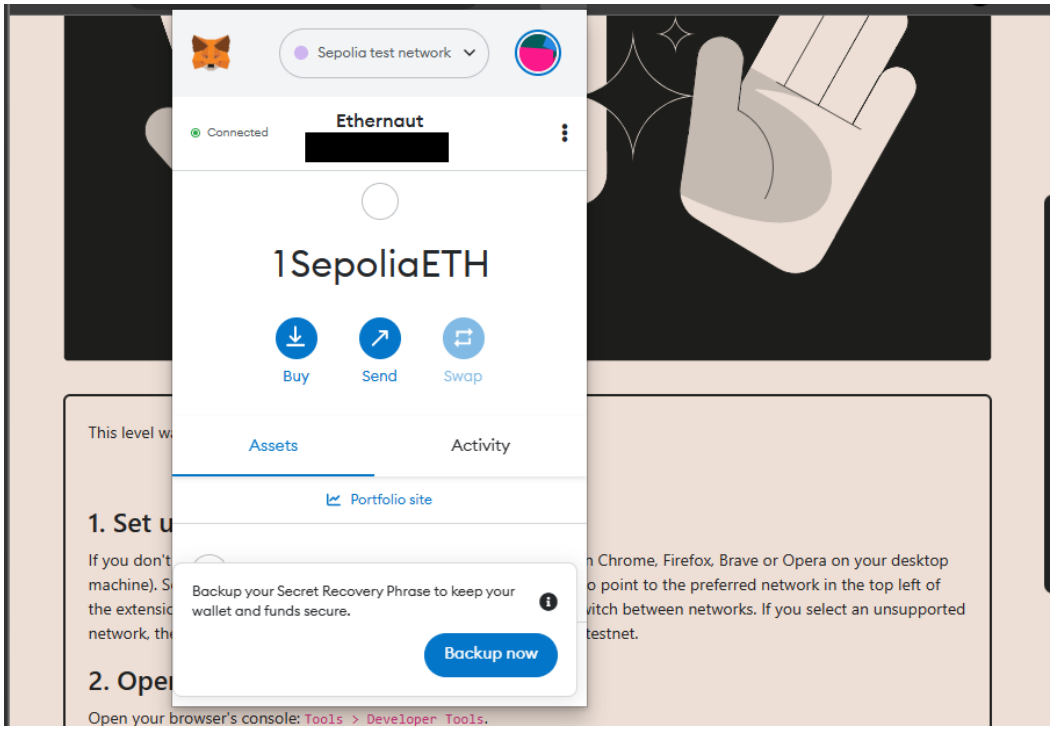
Players may need to use various tools, techniques, and knowledge of blockchain technology to successfully complete the challenges. Playing Ethernaute CTF requires a strong understanding of smart contracts, Solidity programming, and security best practices, making it an engaging and challenging experience for participants looking to test their skills in the field of blockchain security.

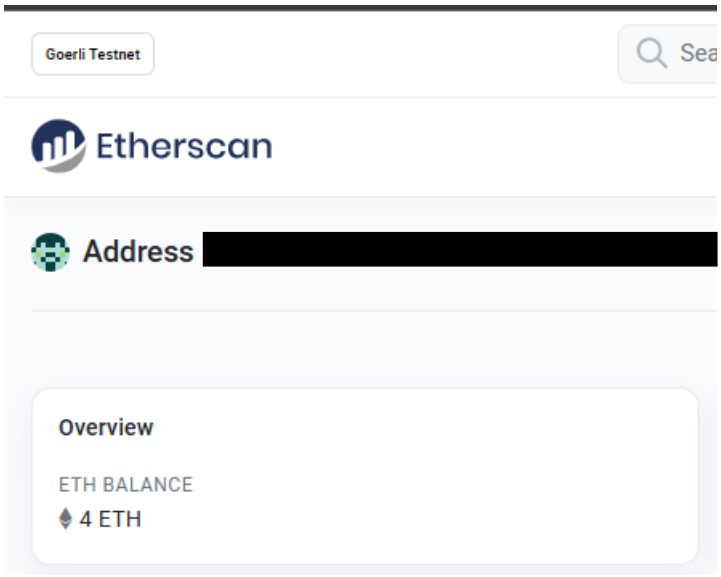
1.Hello Ethernaut – Level 01.



Before getting into hacking Smart contracts, we need configure something. First, we need to configure MetaMask wallet with a valid test network that supported by open zipline also need to fetch some test ETH to that network by using some authorized ETH faucets.

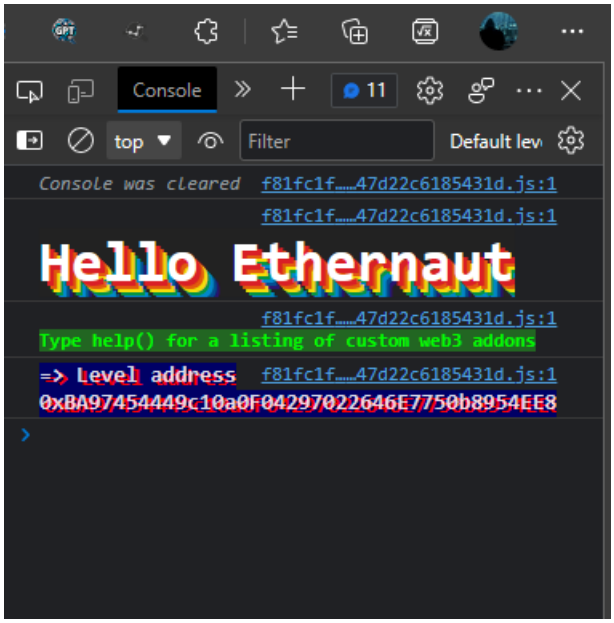
So, I created an account in Meta mask and added a test network called SepoliaETH test network, Goerili test network also added some test ETH to that test network. Because we need those when we are creating an instance from each smart contract, they give to us. To create an instance, we need to pay the gas price.





So, to interact with smart contracts we can use a tool called REMIX IDE. By using that we can simply clone the code and do whatever the analysis needs and finally can update the original smart contract they provided to us to exploit the vulnerability in it. In this Ethernaut CTF competition I'm going to use REMIX IDE as well as Developer tool of the web browser specially the Console to solve each lab.

In belove picture you can see how the actual console is going to look like for Ethernaute CTF.



There are many predefined functions in this challenge. Some of the most important functions are.

1. **Player** – it will return the current player ETH wallet address.

```
> player
< '0x24B5f0A7b8FdB663ccB8F0514426C9623D215875'
```

2. **getBalance(Player) –** will return the current ETH balance of the player.

```
> getBalance(player)
< ▼ Promise {<pending>}
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "0"
```

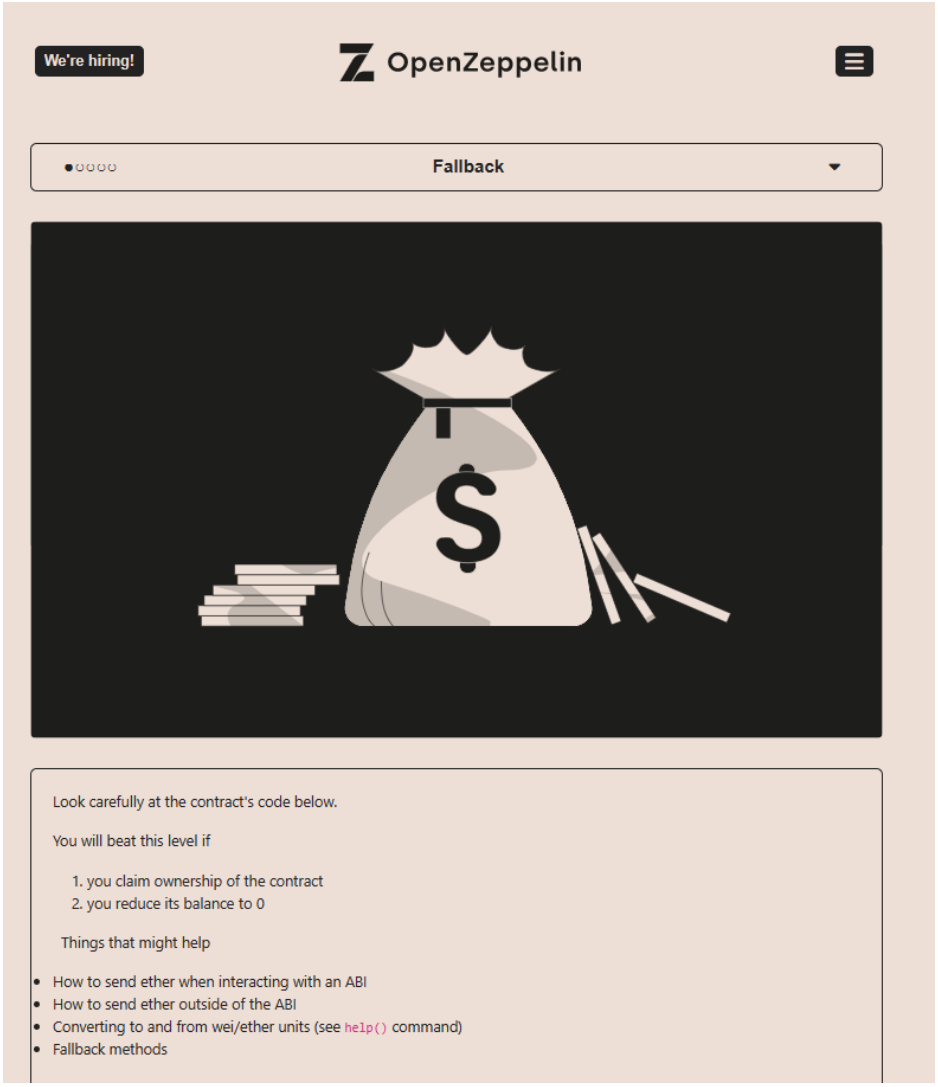
3. **Ethernaut. Owner ()** – Will return the wallet address of the real owner of the Ethernaut contract.

```
> ethernaut.owner()
< ▼ Promise {<pending>, _events: i, emit: ƒ, on:
  ƒ, …}
    ▶ addListener: ƒ (e,t,r)
    ▶ emit: ƒ (e,t,r,n,i,o)
    ▶ listeners: ƒ (e)
    ▶ off: ƒ (e,t,r,n)
    ▶ on: ƒ (e,t,r)
    ▶ once: ƒ (e,t,r)
    ▶ removeAllListeners: ƒ (e)
    ▶ removeListener: ƒ (e,t,r,n)
    ▶ _events: i {}
    ▶ [[Prototype]]: Promise
      [[PromiseState]]: "fulfilled"
      [[PromiseResult]]: "0x09902A56d04a9446601a0d45
```

4. **Help () –** by typing help you can see all available predefined functions for this contract.

So. That's its for this level what we need to get from that level is how to interact with the ethernaut contract and how to create an instance to hack the contract. So, to successfully complete this CTF we need to have a better understanding of the solidity programming language which is used to create smart contracts.

2. Fallback – Level 02.



So, in this level what we need do is first need to take over the ownership of the contract and then need to reduce its balance to 0. Before getting started first need to create an instance of that contract and then need to do whatever changes to the contract that I need to do to take over the ownership of the contract.

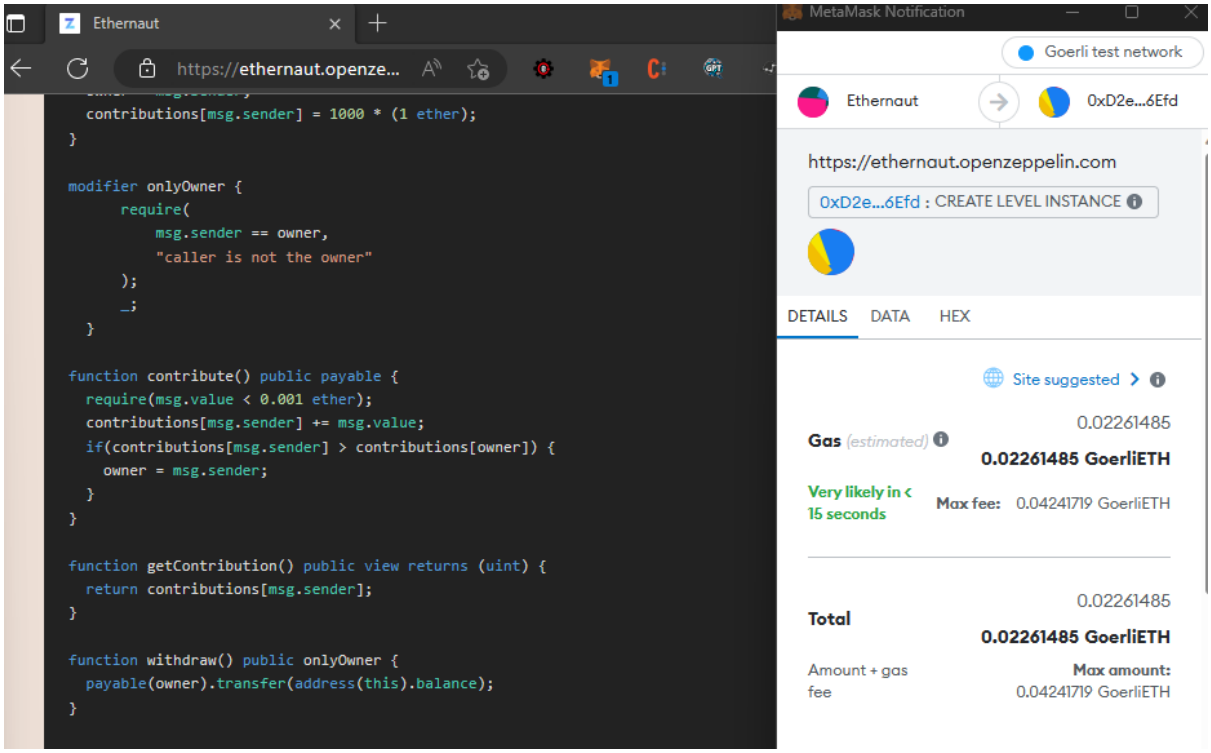


Figure 1: creating an instance from the contract.

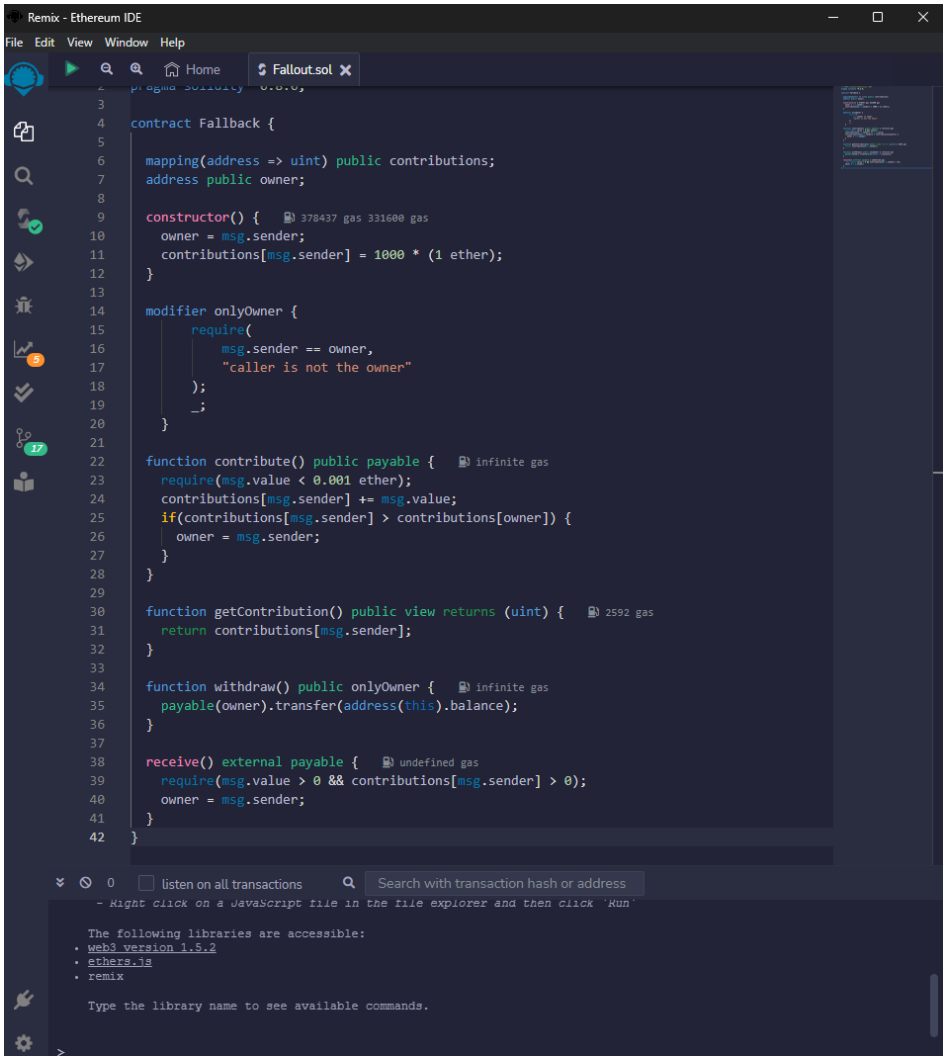
The following is the contract for this level, and we need to hack this contract and need to get ownership of this. And then need to reduce it balance to 0. This .sol file only compiles with version **0.8.0 to 0.9.0** only.

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

contract Fallback {

  mapping(address => uint) public contributions;
  address public owner;

  constructor() {
    owner = msg.sender;
    contributions[msg.sender] = 1000 * (1 ether);
  }

  modifier onlyOwner {
        require(
            msg.sender == owner,
            "caller is not the owner"
        );
        _;
    }

  function contribute() public payable {
    require(msg.value < 0.001 ether);
    contributions[msg.sender] += msg.value;
    if(contributions[msg.sender] > contributions[owner]) {
      owner = msg.sender;
    }
  }

  function getContribution() public view returns (uint) {
    return contributions[msg.sender];
  }

  function withdraw() public onlyOwner {
    payable(owner).transfer(address(this).balance);
  }

  receive() external payable {
    require(msg.value > 0 && contributions[msg.sender] > 0);
    owner = msg.sender;
  }
}
```

To solve the following lab problem, we need to obtain the current player address, contract address and address of the owner. Because while I was analyzing the contract, I figured out only the owner of the contract able to modify call functions and receive ETH.

```solidity
modifier onlyOwner {
      require(
          msg.sender == owner,
          "caller is not the owner"
      );
      _;
  }
```

To solve this problem I'm going to copy past the following contract and going to do whatever the changes need to made in order to gain to ownership of the contract.


Remix - Ethereum IDE
File Edit View Window Help
Home
Fallout.sol
contract Fallback {
mapping(address => uint) public contributions;
address public owner;
constructor() { 378437 gas 331600 gas
owner = msg.sender;
contributions[msg.sender] = 1000 * (1 ether);
}
modifier onlyOwner {
require(
msg.sender == owner,
"caller is not the owner"
);
_;
}
function contribute() public payable { infinite gas
require(msg.value < 0.001 ether);
contributions[msg.sender] += msg.value;
if(contributions[msg.sender] > contributions[owner]) {
owner = msg.sender;
}
}
function getContribution() public view returns (uint) { 2592 gas
return contributions[msg.sender];
}
function withdraw() public onlyOwner { infinite gas
payable(owner).transfer(address(this).balance);
}
receive() external payable { undefined gas
require(msg.value > 0 && contributions[msg.sender] > 0);
owner = msg.sender;
}
}
0
listen on all transactions
Search with transaction hash or address
- Right click on a JavaScript file in the file explorer and then click 'Run'
The following libraries are accessible:
web3 version 1.5.2
ethers.js
remix
Type the library name to see available commands.
>