Smart contracts security

Overview:

Introduction

Blockchain technology has brought about a revolution in various industries by introducing trustless and decentralized systems. However, even within this innovative landscape, smart contracts, which are self-executing agreements on blockchain networks, are not impervious to vulnerabilities and attacks. In this comprehensive guide, we will delve into the realm of smart contract security, shedding light on common vulnerabilities and attacks that can affect these contracts. Our aim is to provide you with a solid foundation in smart contract security, equipping you with essential insights into how these vulnerabilities occur and, more importantly, how to safeguard your smart contracts against them.

Smart contract security is a vast and intricate subject, and this guide serves as your gateway into the expansive realm of securing these digital agreements

Walkthrough

FinTechPro has hired you as a Solidity smart contracts auditor. They have requested that you audit the following smart contract, which is designed to function as a donation fund. It accepts ETH or any native currency of the blockchain it operates on.

The contract's objective is to allow manual withdrawal of funds at the end of each month. These funds are then to be donated to a non-profit organization.

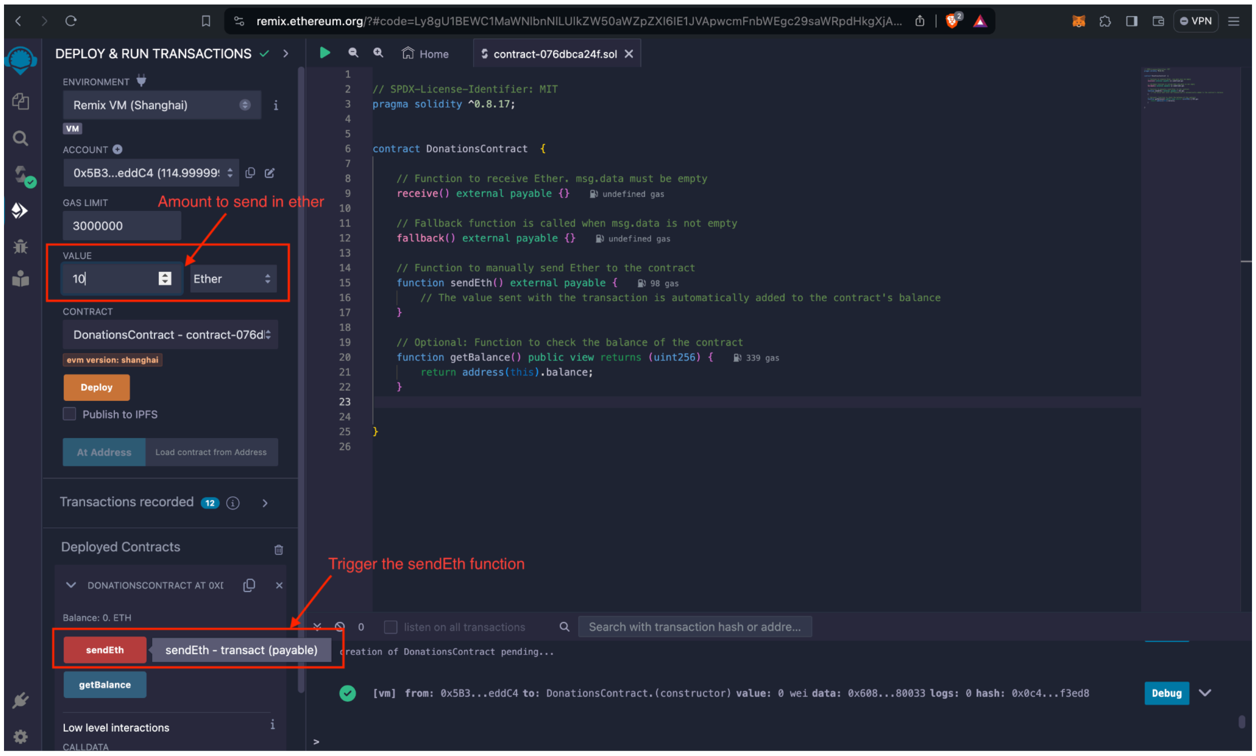
Although it's possible to send ETH directly to the contract's address, an 'ethSend()' function has been included in the contract for ease of use.

Here is the initial contract:

```
C/C++
pragma solidity ^0.8.17;
contract DonationsContract {
  // Function to receive Ether. msg.data must be empty
  receive() external payable {}
  // Fallback function is called when msg.data is not empty
  fallback() external payable {}
  // Function to manually send Ether to the contract
  function sendEth() external payable {
    // The value sent with the transaction is automatically added to the
contract's balance
  }
  // Optional: Function to check the balance of the contract
  function getBalance() public view returns (uint256) {
    return address(this).balance;
  }

}
```

Testing

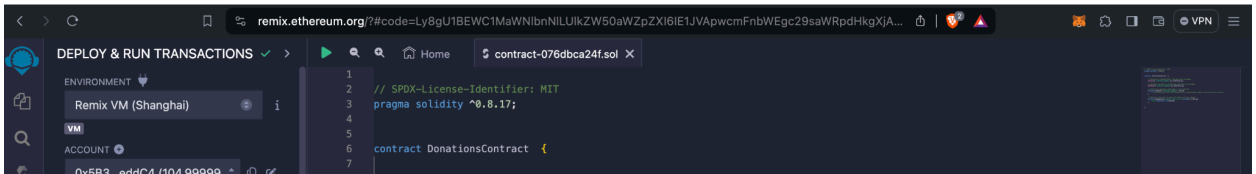
After deploying the contract in the Remix IDE and utilizing its user interface to execute the 'sendEth' function, we observed that it is functioning correctly 🥳!


remix.ethereum.org/?#code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IE1JVApwcmFnbWEgc29saWRpdHkgXjA...
DEPLOY & RUN TRANSACTIONS
Home
contract-076dbca24f.sol
Amount to send in ether
Trigger the sendEth function
Deploy
At Address
Load contract from Address
Transactions recorded 12
Deployed Contracts
sendEth
sendEth - transact (payable)
getBalance
Low level interactions
CALLDATA
listen on all transactions
Search with transaction hash or addre...
Debug
VPN

ENVIRONMENT: Remix VM (Shanghai)
ACCOUNT: 0x5B3...eddC4 (114.99999
GAS LIMIT: 3000000
VALUE: 10 Ether
CONTRACT: DonationsContract - contract-076d
evm version: shanghai
Publish to IPFS
DONATIONSCONTRACT AT 0XI
Balance: 0. ETH

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;


contract DonationsContract  {

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}   undefined gas

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}   undefined gas

    // Function to manually send Ether to the contract
    function sendEth() external payable {   98 gas
        // The value sent with the transaction is automatically added to the contract's balance
    }

    // Optional: Function to check the balance of the contract
    function getBalance() public view returns (uint256) {   339 gas
        return address(this).balance;
    }


}
```

```
creation of DonationsContract pending...

[vm] from: 0x5B3...eddC4 to: DonationsContract.(constructor) value: 0 wei data: 0x608...80033 logs: 0 hash: 0x0c4...f3ed8
>
```


remix.ethereum.org/?#code=Ly8gU1BEWC1MaWNlbnNlLUlkZW50aWZpZXI6IE1JVApwcmFnbWEgc29saWRpdHkgXjA...
DEPLOY & RUN TRANSACTIONS
Home
contract-076dbca24f.sol
VPN

ENVIRONMENT: Remix VM (Shanghai)
ACCOUNT: 0x5B3...eddC4 (104.99999

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;


contract DonationsContract  {
```

Issue 1

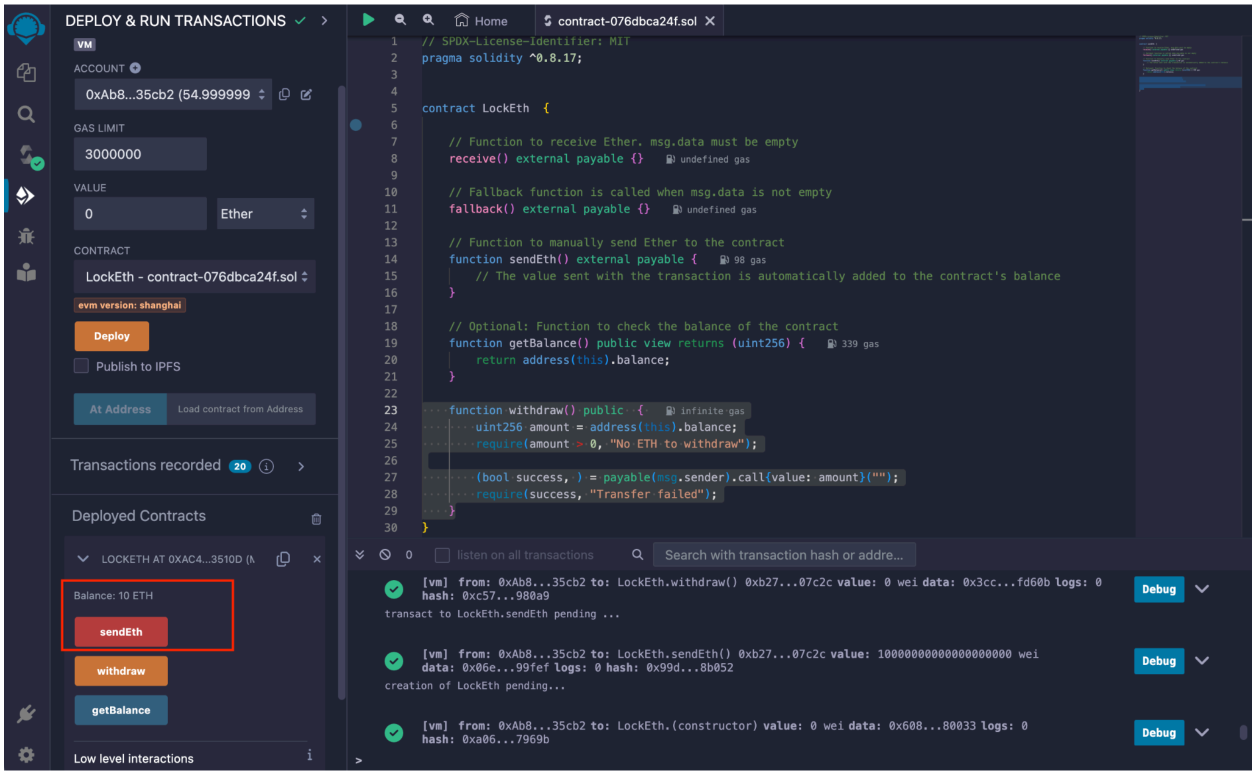
The first issue is that, although we can send ethers to the contract, there's no way to withdraw them. This is a simple fix, but it's crucially important.

Smart contract developers must always ensure the withdrawability of their tokens.

Therefore, we are now adding a 'withdraw' function to the contract, which will enable the withdrawal of the ETH sent to the contract

```
C/C++
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.17;
contract LockEth {

  // Function to receive Ether. msg.data must be empty
  receive() external payable {}
  // Fallback function is called when msg.data is not empty
  fallback() external payable {}
  // Function to manually send Ether to the contract
  function sendEth() external payable {
    // The value sent with the transaction is automatically added to the
contract's balance
  }
  // Optional: Function to check the balance of the contract
  function getBalance() public view returns (uint256) {
    return address(this).balance;
  }
  function withdraw() public {
    uint256 amount = address(this).balance;
    require(amount > 0, "No ETH to withdraw");
    (bool success, ) = payable(msg.sender).call{value: amount}("");
    require(success, "Transfer failed");
  }
}
```

The 'withdraw' function is designed to transfer all Ether from the contract to the address of the person calling the function. It first checks if the contract holds any Ether to send. If there is no Ether, or if the transfer fails, the transaction is reverted. As with the previous step, we first send 10 ETH to the contract.

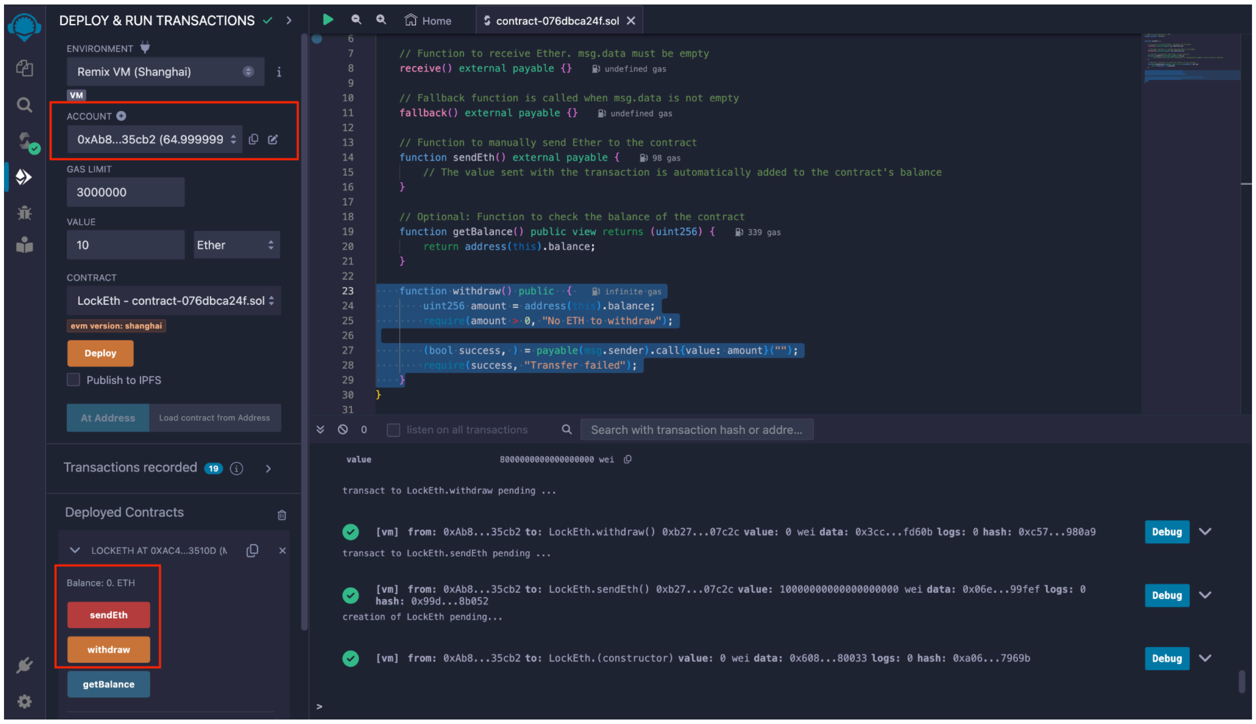


VM
ACCOUNT: 0xAb8...35cb2 (54.999999
GAS LIMIT: 3000000
VALUE: 0 Ether
CONTRACT: LockEth - contract-076dbca24f.sol
evm version: shanghai
Transactions recorded 20
Deployed Contracts
LOCKETH AT 0XAC4...3510D (M
Balance: 10 ETH

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;


contract LockEth  {

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}    undefined gas

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}    undefined gas

    // Function to manually send Ether to the contract
    function sendEth() external payable {    98 gas
        // The value sent with the transaction is automatically added to the contract's balance
    }

    // Optional: Function to check the balance of the contract
    function getBalance() public view returns (uint256) {    339 gas
        return address(this).balance;
    }

    function withdraw() public  {    infinite gas
        uint256 amount = address(this).balance;
        require(amount > 0, "No ETH to withdraw");

        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

```
0    listen on all transactions    Search with transaction hash or addre...

[vm] from: 0xAb8...35cb2 to: LockEth.withdraw() 0xb27...07c2c value: 0 wei data: 0x3cc...fd60b logs: 0
hash: 0xc57...980a9
transact to LockEth.sendEth pending ...

[vm] from: 0xAb8...35cb2 to: LockEth.sendEth() 0xb27...07c2c value: 10000000000000000000 wei
data: 0x06e...99fef logs: 0 hash: 0x99d...8b052
creation of LockEth pending...

[vm] from: 0xAb8...35cb2 to: LockEth.(constructor) value: 0 wei data: 0x608...80033 logs: 0
hash: 0xa06...7969b
>
```

Next, we test our withdraw function by examining the account balance and using the 'getBalance' function. It appears to be working correctly!



ENVIRONMENT: Remix VM (Shanghai)
VM
ACCOUNT: 0xAb8...35cb2 (64.999999
GAS LIMIT: 3000000
VALUE: 10 Ether
CONTRACT: LockEth - contract-076dbca24f.sol
evm version: shanghai
Transactions recorded 19
Deployed Contracts
LOCKETH AT 0XAC4...3510D (M
Balance: 0. ETH

```

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}    undefined gas

    // Fallback function is called when msg.data is not empty
    fallback() external payable {}    undefined gas

    // Function to manually send Ether to the contract
    function sendEth() external payable {    98 gas
        // The value sent with the transaction is automatically added to the contract's balance
    }

    // Optional: Function to check the balance of the contract
    function getBalance() public view returns (uint256) {    339 gas
        return address(this).balance;
    }

    function withdraw() public  {    infinite gas
        uint256 amount = address(this).balance;
        require(amount > 0, "No ETH to withdraw");

        (bool success, ) = payable(msg.sender).call{value: amount}("");
        require(success, "Transfer failed");
    }
}

```

```
0    listen on all transactions    Search with transaction hash or addre...

value                    8000000000000000000 wei

transact to LockEth.withdraw pending ...

[vm] from: 0xAb8...35cb2 to: LockEth.withdraw() 0xb27...07c2c value: 0 wei data: 0x3cc...fd60b logs: 0 hash: 0xc57...980a9
transact to LockEth.sendEth pending ...

[vm] from: 0xAb8...35cb2 to: LockEth.sendEth() 0xb27...07c2c value: 10000000000000000000 wei data: 0x06e...99fef logs: 0
hash: 0x99d...8b052
creation of LockEth pending...

[vm] from: 0xAb8...35cb2 to: LockEth.(constructor) value: 0 wei data: 0x608...80033 logs: 0 hash: 0xa06...7969b

>
```

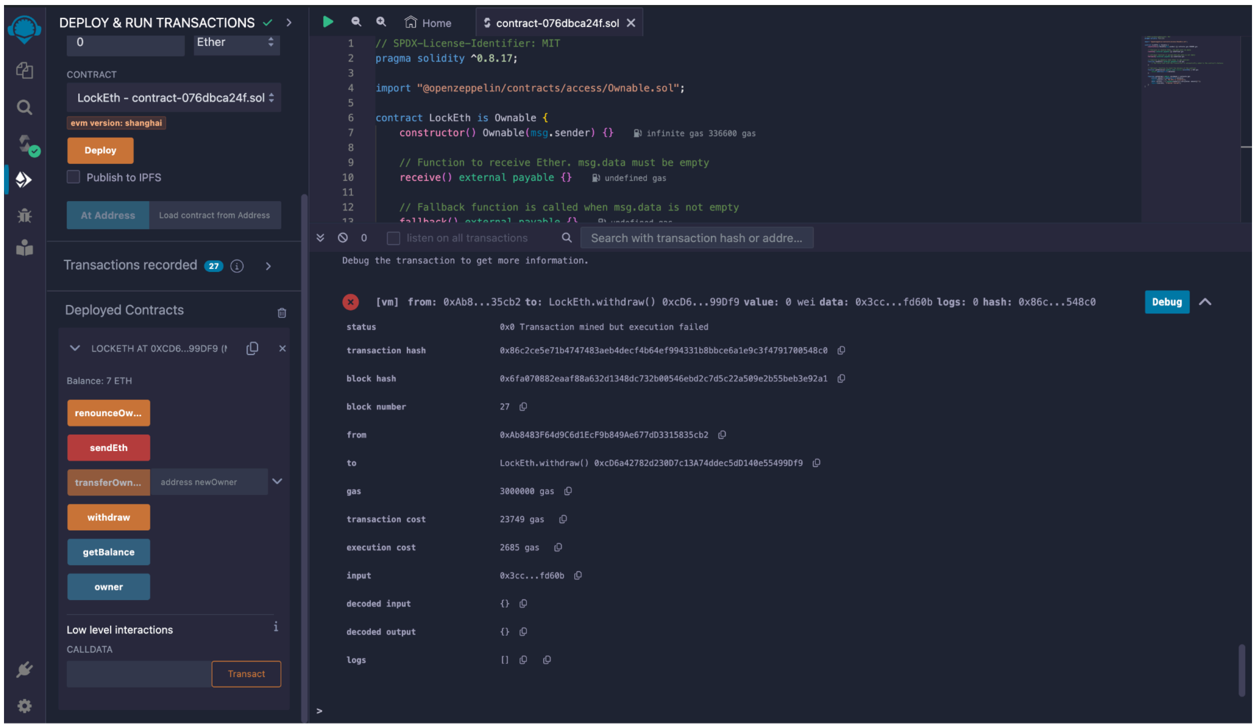
Issue 2:

After confirming that our withdraw function operates correctly, we identified a significant issue: anyone can withdraw the funds from our smart contract. To resolve this, we need to implement access control.

To achieve this, we are utilizing the OpenZeppelin 'Ownable' library. We will set up the 'onlyOwner' modifier in the withdraw function and change 'msg.sender' to 'owner()'. This modification ensures that only the owner (the deployer of the contract) can withdraw the funds.

```
C/C++
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;
import "@openzeppelin/contracts/access/Ownable.sol";

contract LockEth is Ownable {

  constructor() Ownable(msg.sender) {}
  // Function to receive Ether. msg.data must be empty
  receive() external payable {}
  // Fallback function is called when msg.data is not empty
  fallback() external payable {}
  // Function to manually send Ether to the contract
  function sendEth() external payable {
    // The value sent with the transaction is automatically added to the
contract's balance
  }
  // Optional: Function to check the balance of the contract
  function getBalance() public view returns (uint256) {
    return address(this).balance;
  }
  function withdraw() public onlyOwner {
    uint256 amount = address(this).balance;
    require(amount > 0, "No ETH to withdraw");
    (bool success, ) = payable(owner()).call{value: amount}("");
    require(success, "Transfer failed");
  }
}
```

Now that we have set up access control, we test the withdraw function using an account different from the owner's. As a result, we receive the following error in the console:



DEPLOY & RUN TRANSACTIONS

VALUE: 0 Ether

CONTRACT: LockEth - contract-076dbca24f.sol

evm version: shanghai

Transactions recorded 27

Deployed Contracts

LOCKETH AT 0XCD6...99DF9

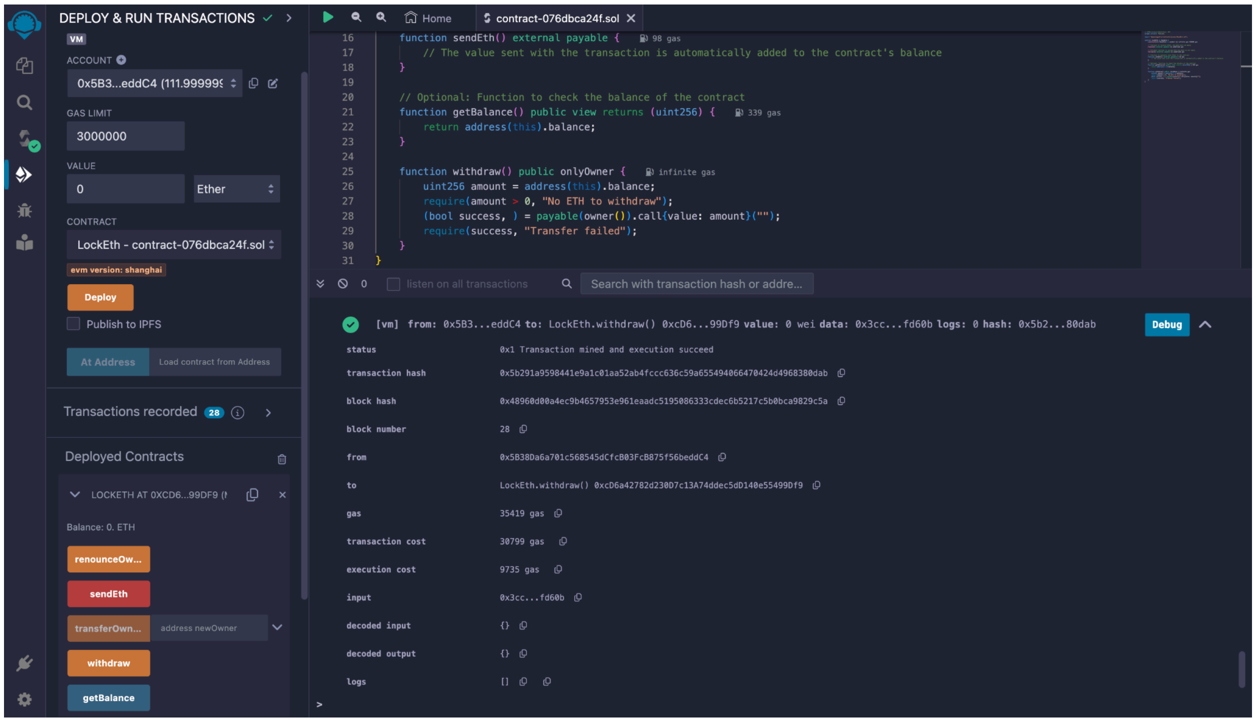
Balance: 7 ETH

contract-076dbca24f.sol

```solidity
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.17;

import "@openzeppelin/contracts/access/Ownable.sol";

contract LockEth is Ownable {
    constructor() Ownable(msg.sender) {}    infinite gas 336600 gas

    // Function to receive Ether. msg.data must be empty
    receive() external payable {}    undefined gas

    // Fallback function is called when msg.data is not empty
```

```
Debug the transaction to get more information.

[vm] from: 0xAb8...35cb2 to: LockEth.withdraw() 0xcD6...99Df9 value: 0 wei data: 0x3cc...fd60b logs: 0 hash: 0x86c...548c0
status               0x0 Transaction mined but execution failed
transaction hash     0x86c2ce5e71b4747483aeb4decf4b64ef994331b8bbce6a1e9c3f4791700548c0
block hash           0x6fa070882eaaf88a632d1348dc732b00546ebd2c7d5c22a509e2b55beb3e92a1
block number         27
from                 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to                   LockEth.withdraw() 0xcD6a42782d230D7c13A74ddec5dD140e55499Df9
gas                  3000000 gas
transaction cost     23749 gas
execution cost       2685 gas
input                0x3cc...fd60b
decoded input        {}
decoded output       {}
logs                 []
```

And when we attempt to withdraw using the owner's account, it works successfully!



DEPLOY & RUN TRANSACTIONS

VM

ACCOUNT: 0x5B3...eddC4 (111.99999

GAS LIMIT: 3000000

VALUE: 0 Ether

CONTRACT: LockEth - contract-076dbca24f.sol

evm version: shanghai

Transactions recorded 28

Deployed Contracts

LOCKETH AT 0XCD6...99DF9

Balance: 0. ETH

contract-076dbca24f.sol

```solidity
    function sendEth() external payable {    98 gas
        // The value sent with the transaction is automatically added to the contract's balance
    }

    // Optional: Function to check the balance of the contract
    function getBalance() public view returns (uint256) {    339 gas
        return address(this).balance;
    }

    function withdraw() public onlyOwner {    infinite gas
        uint256 amount = address(this).balance;
        require(amount > 0, "No ETH to withdraw");
        (bool success, ) = payable(owner()).call{value: amount}("");
        require(success, "Transfer failed");
    }
}
```

```
[vm] from: 0x5B3...eddC4 to: LockEth.withdraw() 0xcD6...99Df9 value: 0 wei data: 0x3cc...fd60b logs: 0 hash: 0x5b2...80dab
status               0x1 Transaction mined and execution succeed
transaction hash     0x5b291a9598441e9a1c01aa52ab4fccc636c59a655494066470424d4968380dab
block hash           0x48960d00a4ec9b4657953e961eaadc5195086333cdec6b5217c5b0bca9829c5a
block number         28
from                 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to                   LockEth.withdraw() 0xcD6a42782d230D7c13A74ddec5dD140e55499Df9
gas                  35419 gas
transaction cost     30799 gas
execution cost       9735 gas
input                0x3cc...fd60b
decoded input        {}
decoded output       {}
logs                 []
```

Common Vulnerabilities and Attacks

1. Reentrancy Attack

This occurs when a contract makes an external call to another untrusted contract and the external contract calls back into the calling contract before the first execution finishes.

Example:

Exploitable

In this exploitable example, the contract updates the balance after the withdrawal.
An attacker can exploit this by repeatedly calling the withdraw function in a fallback function, draining the contract's funds.

```JavaScript
contract ExploitableWithdraw {
    mapping(address => uint256) public balances;
    function withdraw(uint256 amount) external {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Withdrawal failed");
        balances[msg.sender] -= amount;
    }
}
```

Secure

The secure example uses the ReentrancyGuard library,
preventing reentrancy attacks by ensuring state changes (balance update) happen before the external call. The nonReentrant library ensures that a function can't be re-entered while it's still executing.

```C/C++
contract SecureWithdraw is ReentrancyGuard {
    mapping(address => uint256) public balances;
    function withdraw(uint256 amount) external nonReentrant {
        require(balances[msg.sender] >= amount, "Insufficient balance");
        balances[msg.sender] -= amount;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Withdrawal failed");
    }
}
```

2.Access Control

These vulnerabilities occur when unauthorized users gain access to privileged functions or data within a smart contract. For example, if a contract lacks proper access controls, any user can call functions meant for administrators or owners, potentially leading to unauthorized modifications, theft, or other malicious actions. Proper access control mechanisms, such as using modifiers or checks, are essential to restrict access to sensitive contract functionality and data, preventing unauthorized actions.

Exploitable

This exploitable example, there's no access control implemented, allowing any user to call the restrictedFunction

```
C/C++
contract ExploitableAccessControl {
    uint256 private secretValue;

    function restrictedFunction(uint256 _value) external {
        secretValue = _value;
    }

    function getSecretValue() external view returns (uint256) {
        return secretValue;
    }
}
```

Secure

The secure example uses the Ownable pattern to restrict access to the restrictedFunction, ensuring that only the contract owner can call it.

```
C/C++
import "@openzeppelin/contracts/access/Ownable.sol";

contract SecureAccessControl is Ownable {
    uint256 private secretValue;

    function setSecretValue(uint256 _value) external onlyOwner {
        secretValue = _value;
    }

    function getSecretValue() external view returns (uint256) {
        require(msg.sender == owner(), "Unauthorized");
        return secretValue;
    }
}
```

3. Denial of service

DoS attacks target the availability and functionality of a smart contract by overloading it with malicious actions. Attackers may repeatedly call functions that consume excessive gas, causing the contract to run out of gas and become unusable. This can disrupt the entire blockchain network and affect other users' transactions. Preventing DoS attacks requires efficient gas management, rate limiting, and careful design to ensure that contract functions cannot be excessively abused.

Pull Payment Pattern, In this exploitable example, the contract pushes payments to recipients, which can lead to denial-of-service attacks if a malicious recipient prevents the contract from sending payments.

Exploitable

```
C/C++
contract ExploitablePushPayment {
    function sendPayment(address payable recipient, uint256 amount) external {
        (bool success, ) = recipient.call{value: amount}("");
        require(success, "Payment failed");
    }
}
```

Secure

Pull Payment Pattern,The secure example uses the pull payment pattern,
allowing recipients to withdraw their funds when they choose, reducing the risk of potential denial-of-service attacks.

```
C/C++
contract SecurePullPayment {
    mapping(address => uint256) public pendingWithdrawals;

    function deposit(address recipient, uint256 amount) external {
        pendingWithdrawals[recipient] += amount;
    }

    function withdraw() external {
        uint256 amount = pendingWithdrawals[msg.sender];
        require(amount > 0, "No pending withdrawals");
        pendingWithdrawals[msg.sender] = 0;
        (bool success, ) = msg.sender.call{value: amount}("");
        require(success, "Withdrawal failed");
    }
}
```

4.Arithmetic Vulnerabilities

Arithmetic vulnerabilities are a critical concern in smart contract security. They occur when integer arithmetic operations are not properly validated, potentially leading to integer overflows and underflows. In Solidity, the introduction of automatic checks for these vulnerabilities starting from version 0.8.0 has significantly improved security. In this guide, we'll explore the risks posed by arithmetic vulnerabilities and how the latest Solidity updates enhance contract safety.

Exploitable

In this exploitable example, the contract uses unchecked arithmetic operations,
which can lead to integer overflows and underflows.

```
C/C++
contract ExploitableArithmetic {
    function unsafeAdd(uint256 a, uint256 b) public pure returns (uint256) {
        return a + b;
    }
}
```

Secure

SafeMath is used to perform safe arithmetic operations. It ensures that adding a and b doesn't result in integer overflow or underflow, preventing unexpected and potentially exploitable behavior. The add function checks for these conditions and reverts the transaction if they occur, making arithmetic operations secure.

```
C/C++
import "@openzeppelin/contracts/utils/math/SafeMath.sol";

contract SecureArithmetic {
    using SafeMath for uint256;

    function safeAdd(uint256 a, uint256 b) public pure returns (uint256) {
        return a.add(b);
    }
}
```

5. Accessing Private Data

On public blockchains, all the data on a smart contract can be read, including data intended to be private.

The following code demonstrates how you can read “private” data off any public blockchain

```
C/C++
```

Preventative Techniques

Don't store sensitive unencrypted information on the public blockchain.

Examples famous of hacks

- **Euler Finance Hack (2023)**: This incident involved a flash loan attack, leading to a staggering loss of $197 million. The vulnerability was exploited to drain funds in a sophisticated manner.

  https://medium.com/buildbear/a-comprehensive-analysis-of-euler-finances-196-million-flash-loan-exploit-and-a-step-by-step-guide-691a31f26452

- **Mixin Breach (2023)**: Mixin suffered a significant setback when their cloud service provider was breached, resulting in a loss of $200 million. This hack underscored the risks associated with third-party service providers in the blockchain ecosystem.

  https://coinpedia.org/research-report/crypto-security-breaches-hacks-of-2023-a-detailed-analysis/

- **CoinEx Hack (2023)**: The CoinEx platform faced a severe security breach when compromised private keys led to the theft of over $70 million in various tokens. This incident highlighted the critical importance of secure key management.
- **HECO Bridge and HTX Hack (2023)**: A sophisticated attack on the HECO bridge resulted in a loss of $120 million, showcasing vulnerabilities in cross-chain bridges.
- **Atomic Wallet Hack (2023)**: In a massive cryptocurrency heist, the Atomic Wallet was compromised, leading to the loss of millions of dollars. This hack brought to light the security challenges in wallet infrastructure.
- **The DAO Hack (2016)**: One of the most infamous incidents in the blockchain world, the DAO hack resulted in the theft of 3.6 million Ether, worth around $50 million at the time. This exploit was due to vulnerabilities in the DAO's smart contract code and led to a hard fork in the Ethereum network, ultimately resulting in the split between Ethereum (ETH) and Ethereum Classic (ETC).
- **Terra Virtua (2022)**: This attack led to a loss of $32 million, emphasizing the ongoing threats to virtual asset platforms.
- **Cream Finance (2021)**: Resulting in a loss of $130 million, this hack was a significant blow to the DeFi sector.
- **Poly Network (2021)**: A massive hack involving $600 million, which was notably returned by the hacker, showcasing a unique twist in the world of crypto heists.

Additional resources

To discover more vulnerabilities in depth we recommend checking

1.https://solidity-by-example.org/hacks/self-destruct/
2. https://twitter.com/i/lists/1620633132524503046?s=20
3. https://web3securitydao.xyz/collaborating/resources