uPort

uPort Documentation

Launch A Decentralized Identity Application using the Developer Friendly uPort/React Truffle Box

ENS website
ENS documentation

IPFS

IPFS Demo

IPFS Docs

An introduction to IPFS - ConsenSys

Writing Upgradable Contracts in solidity

Software Engineering techniques - ConsenSys

Upgradable smart contracts. What we've learned building Insurance on a Blockchain
Summary of Ethereum Upgradeable Smart Contract R&D

0x Protocol Wiki

Oraclize remix demo

Oraclize documentation

This document is the adaptation of the presentation slides into a more easily readable format for reading.

There are several design patterns and best practices that have been developed.

Fail early and fail loud

```
//Bad code, do not emulate

function silentFailIfZero(uint num) public view returns (uint){

    if(num != 0){

        return num;

    }

}
```

This function will fail without throwing an exception. This is a bad practice because it is not immediately clear whether the function executed properly or not.

```
function throwsErrorIfZero(uint num) public view returns (uint){

    require(num != 0);

    return num;

}
```

This function checks the condition required for execution as early as possible in the function body and throws an exception if the condition is not met. This is a good practice to reduce unnecessary code execution in the event that an exception will be thrown.

Restricting Access

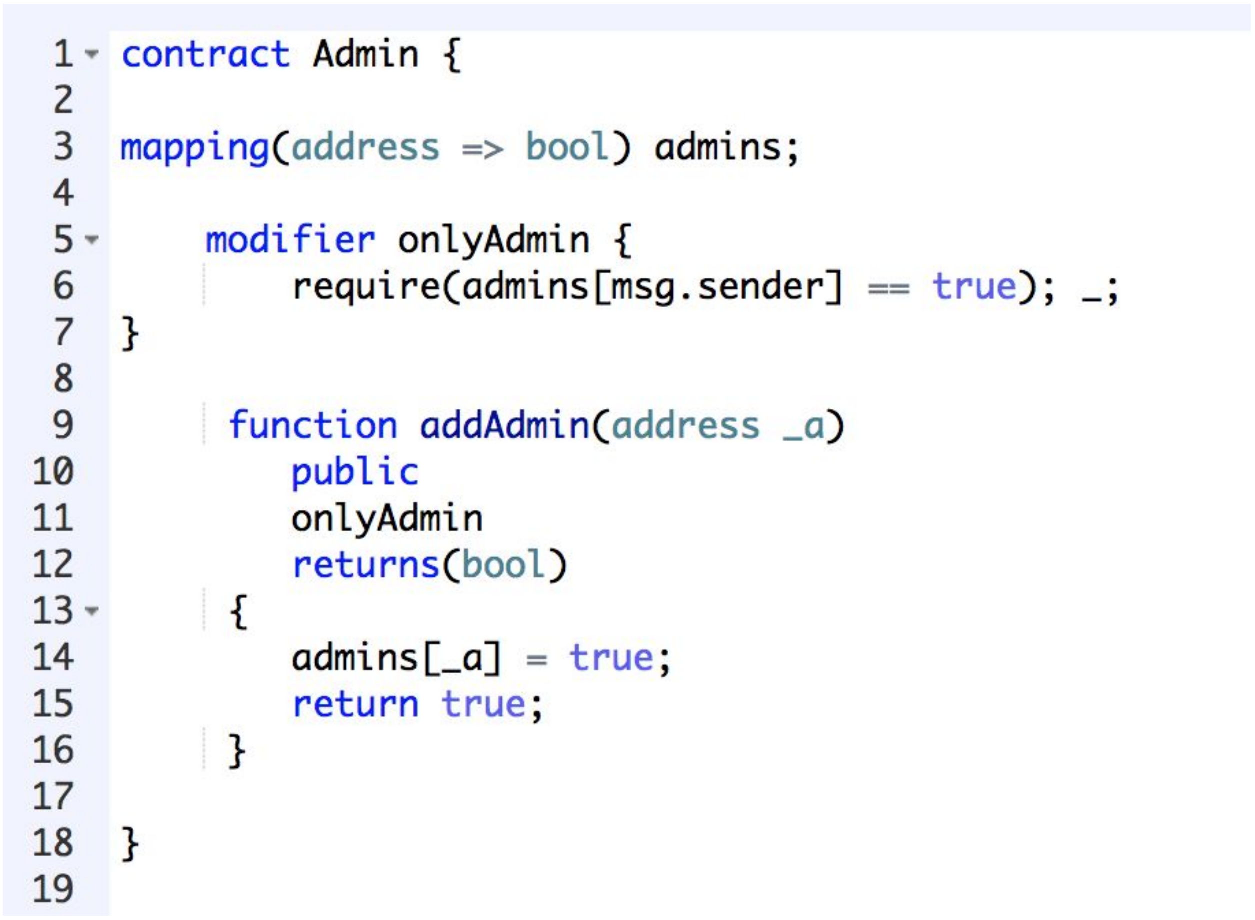
You cannot prevent people or computer programs from reading your contracts' state. The state is publicly available information.

You can restrict other contracts' access to the state by making state variables private.

```
contract C1 {

uint private internalNum;

}
```

You can restrict function access so that only specific addresses are permitted to execute functions.

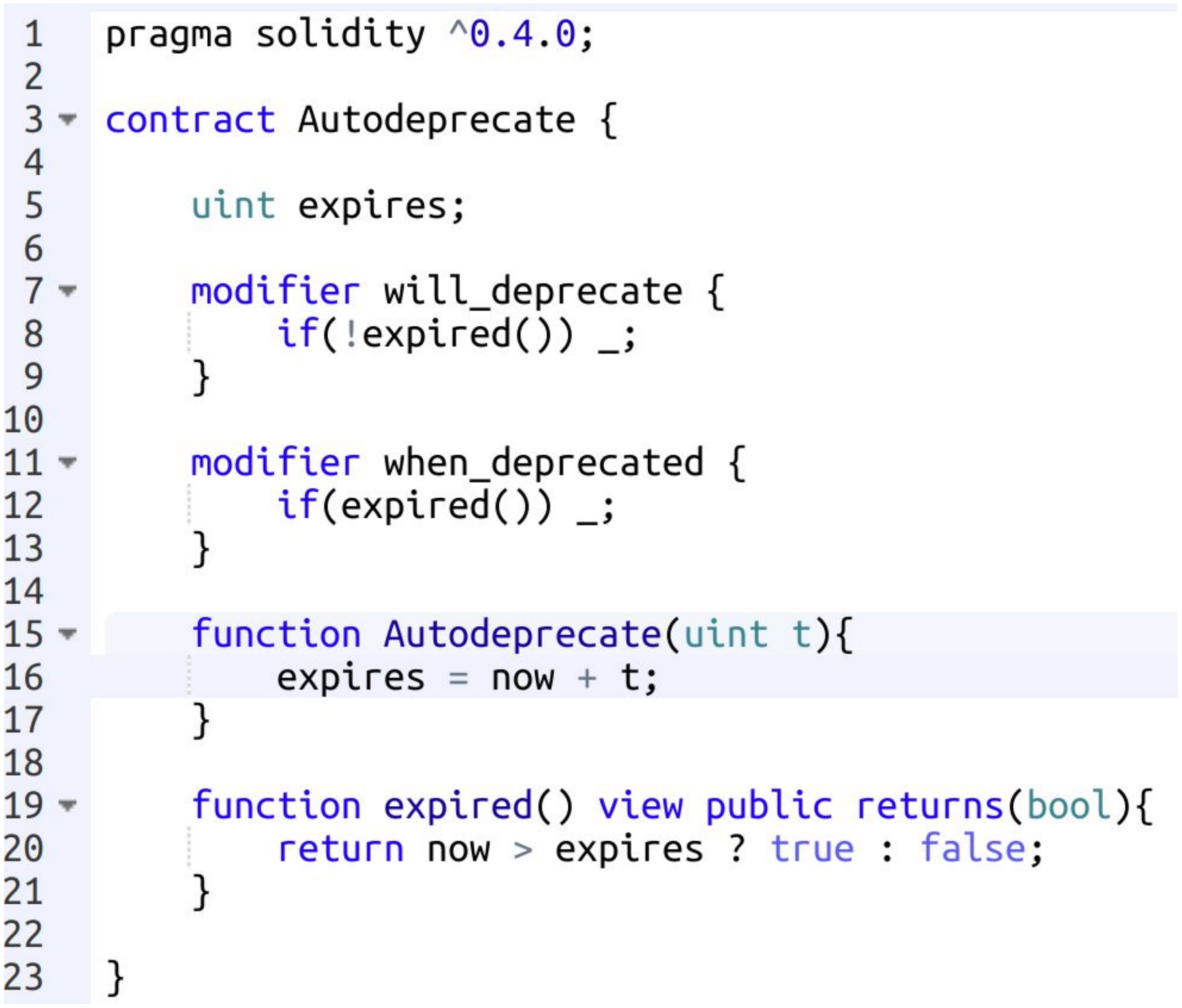
This is useful for allowing only designated users, or other contracts to access administrative methods, such as changing ownership of a contract, implementing an upgrade or stopping the contract.



```
contract Admin {

mapping(address => bool) admins;

    modifier onlyAdmin {
        require(admins[msg.sender] == true); _;
}

     function addAdmin(address _a)
        public
        onlyAdmin
        returns(bool)
     {
        admins[_a] = true;
        return true;
     }

}

```

It can be useful to restrict function access to owners, a more general admin class or to any stakeholder in the system.

Auto Deprecation



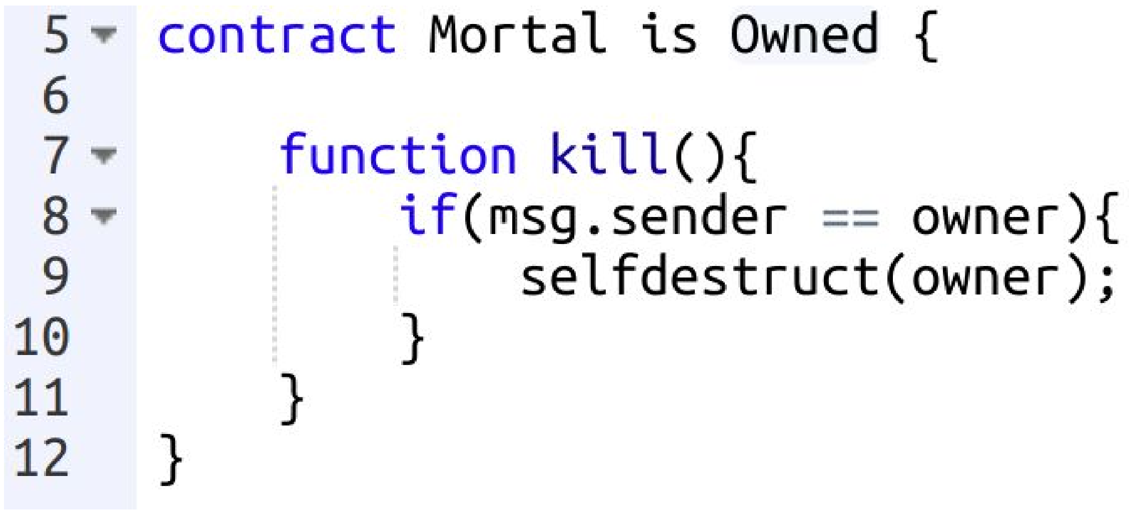
```
pragma solidity ^0.4.0;

contract Autodeprecate {

    uint expires;

    modifier will_deprecate {
        if(!expired()) _;
    }

    modifier when_deprecated {
        if(expired()) _;
    }

    function Autodeprecate(uint t){
        expires = now + t;
    }

    function expired() view public returns(bool){
        return now > expires ? true : false;
    }

}
```

The auto deprecation design pattern is a useful strategy for closing contracts that should expire after a certain amount of time.

This can be useful when running alpha or beta testing for your smart contracts.

Remember that using timestamps such as the now keyword are subject to manipulation by the block miners in a 30-second window.

Mortal



```
contract Mortal is Owned {

    function kill(){
        if(msg.sender == owner){
            selfdestruct(owner);
        }
    }
}
```

Implementing the mortal design pattern means including the ability to destroy the contract and remove it from the blockchain.

You can destroy a contract using the **selfdestruct** keyword. The function to do it is often called kill.

It takes one parameter which is the address that will receive all of the funds that the contract currently holds.

As an irreversible action, restricting access to this function is important.

Pull over Push Payments

Navigate to this fund splitter contract on github.

There are a few key takeaways from this contract. There is a separation of function logic. The split() function handles the accounting and divides the msg.value sent with the transaction. Another function, withdraw(), allows accounts to transfer their balance from the contract to their account.

This pattern is also called the withdrawal pattern. It protects against re-entrancy and denial of service attacks that we will cover in the next lesson.

Circuit Breaker

Circuit Breakers are design patterns that allow contract functionality to be stopped. This would be desirable in situations where there is a live contract where a bug has been detected. Freezing the contract would be beneficial for reducing harm before a fix can be implemented.

Circuit breaker contracts can be set up to permit certain functions in certain situations. For example, if you are implementing a withdrawal pattern, you might want to stop people from depositing funds into the contract if a bug has been detected, while still allowing accounts with balances to withdraw their funds.

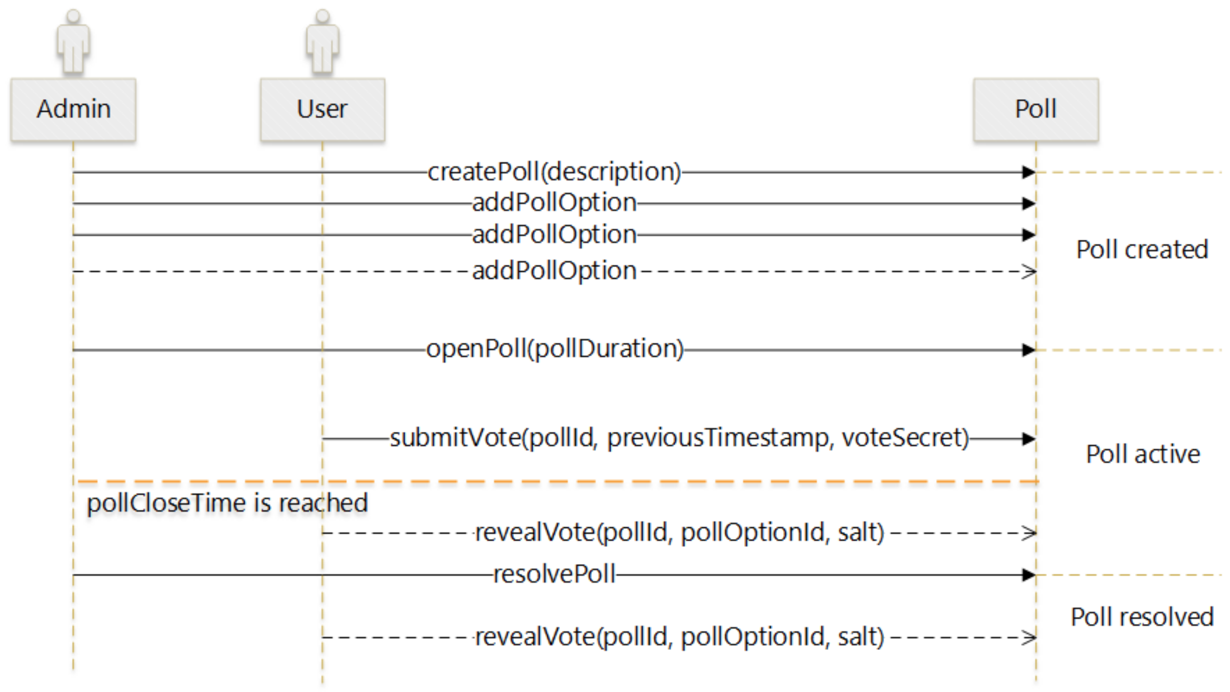
```
contract CircuitBreaker {

     bool public stopped = false;

     modifier stopInEmergency { require(!stopped); _; }

     modifier onlyInEmergency { require(stopped); _; }

     function deposit() stopInEmergency public { … }

     function withdraw() onlyInEmergency public { … }

}
```

In a situation such as this, you would also want to restrict access to the accounts that can modify the **stopped** state variable, maybe to the contract owner (such as multisig wallet) or a set of admins.

State Machine

Contracts often act as a state machine, which means that they have certain **stages** in which they behave differently or in which different functions can be called. A function call often ends a stage and transitions the contract into the next stage (especially if the contract models **interaction**). It is also common that some stages are automatically reached at a certain point in **time**.

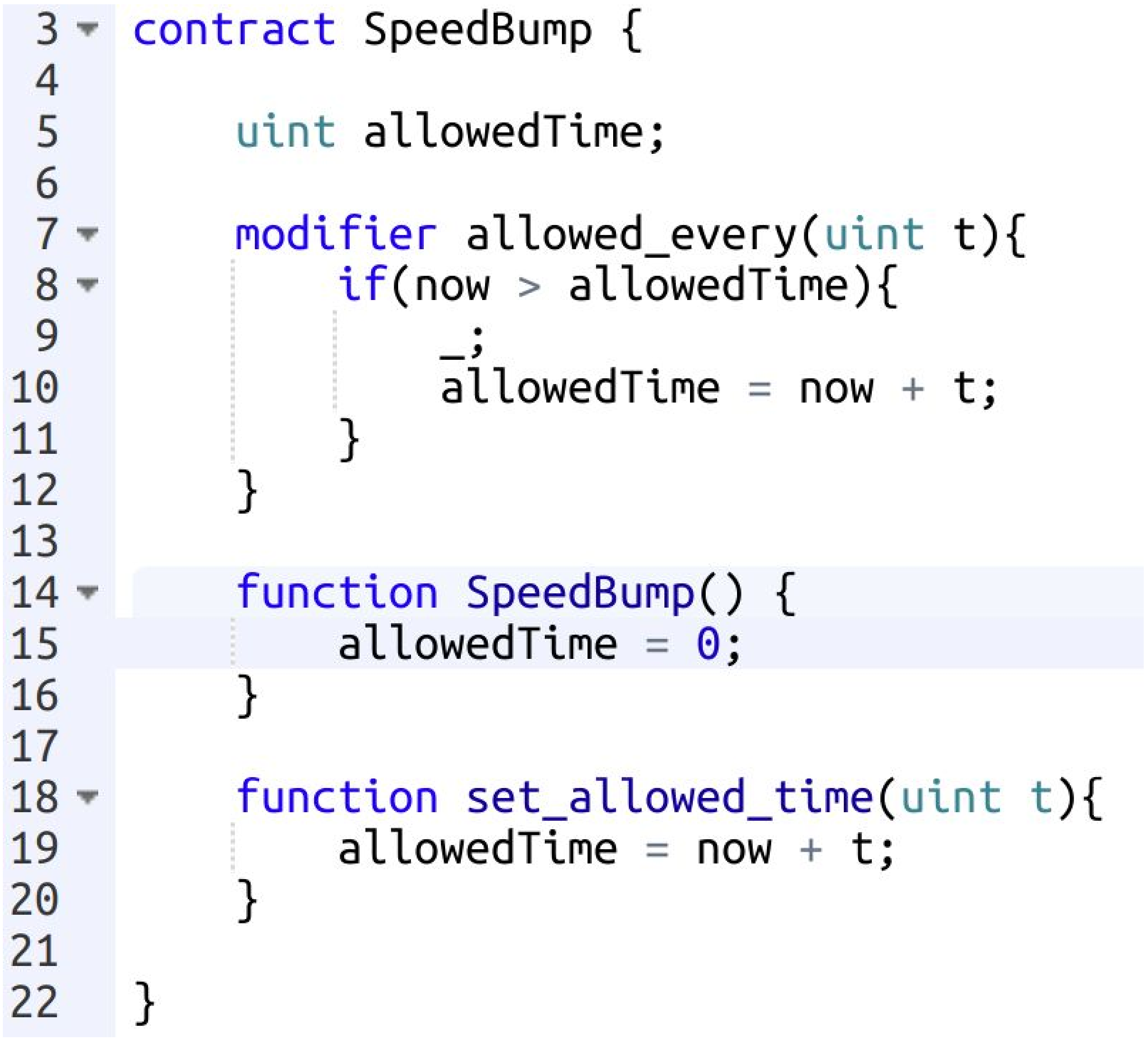
The Colony token weighted voting protocol implemented this design pattern to manage the poll state.



Admins can only add poll options in the poll creation stage. Votes can only be submitted when the poll was active. The poll can only be resolved after the poll close time has been reached.

Speed Bump

Speed bumps slow down actions so that if malicious actions occur, there is time to recover.



```
contract SpeedBump {

    uint allowedTime;

    modifier allowed_every(uint t){
        if(now > allowedTime){
            _;
            allowedTime = now + t;
        }
    }

    function SpeedBump() {
        allowedTime = 0;
    }

    function set_allowed_time(uint t){
        allowedTime = now + t;
    }

}
```

For example, The DAO required 27 days between a successful request to split the DAO and the ability to do so. This ensured the funds were kept within the contract, increasing the likelihood of recovery.

Useful links:

Solidity docs - Common Design Patterns

Design Patterns

Pull over push payments - github gist

****

This document is the adaptation of the presentation slides into a more easily readable format for reading.

The first set of dangers that we are going to look at fall in the class of race conditions.

Reentrancy attacks can be problematic because calling external contracts passes control flow to them. The called contract may end up calling the smart contract function again in a recursive manner.

```
// INSECURE
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    require(msg.sender.call.value(amountToWithdraw)()); // At this point, the
caller's code is executed, and can call withdrawBalance again
    userBalances[msg.sender] = 0;
}
```

This is the type of attack that destroyed the DAO in mid 2016.

If you can’t remove the external call, the next simplest way to prevent this attack is to do the internal work before making the external function call.

```
mapping (address => uint) private userBalances;

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    userBalances[msg.sender] = 0;
    require(msg.sender.call.value(amountToWithdraw)()); // The user's balance
is already 0, so future invocations won't withdraw anything
}
```

Or to use the withdrawal design pattern and separate the contract accounting logic and the transfer logic.

Another thing to be aware of is potential cross function race conditions. This can be problematic if your contract has multiple functions that modify the same state.

```
// INSECURE
mapping (address => uint) private userBalances;

function transfer(address to, uint amount) {
    if (userBalances[msg.sender] >= amount) {
       userBalances[to] += amount;
       userBalances[msg.sender] -= amount;
    }
}

function withdrawBalance() public {
    uint amountToWithdraw = userBalances[msg.sender];
    require(msg.sender.call.value(amountToWithdraw)()); // At this point, the
caller's code is executed, and can call transfer()
    userBalances[msg.sender] = 0;
}
```

Ask yourself, **if multiple contract functions are called, what happens?**

There are several ways to mitigate these problems.

It is generally a good idea to handle your internal contract state changes before calling external contracts, such as in the withdrawal design pattern. Use battle tested design patterns and learn from other people’s mistakes and heed their advice.

A more complex solution could implement mutual exclusion, or a mutex. This allows you to lock a state and only allow changes by the owner of the lock.

You can dig deeper into known attacks such as these here.

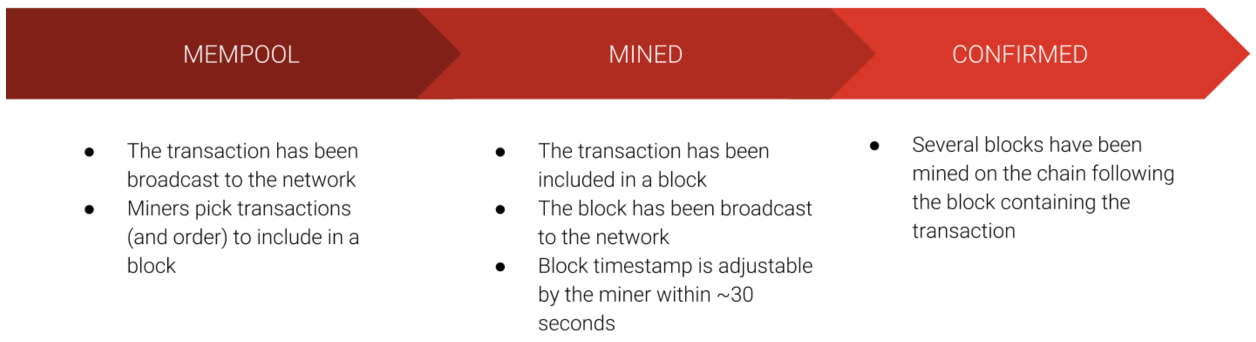
Transaction Ordering and Timestamp Dependence

The previous examples of race conditions involved an attacker executing malicious code in a single transaction. Here we focus on how transactions are included in the blockchain and considerations around the process.

Transactions that are broadcast to the network but have not yet been included in a block are in the mempool.

Miners choose the order in which to include transactions from the mempool into a block that they are mining.

Also, since transactions are in the mempool before they make it into a block, anyone can know what transactions are about to occur on the network.


MEMPOOL
MINED
CONFIRMED
● The transaction has been broadcast to the network
● Miners pick transactions (and order) to include in a block
● The transaction has been included in a block
● The block has been broadcast to the network
● Block timestamp is adjustable by the miner within ~30 seconds
● Several blocks have been mined on the chain following the block containing the transaction

This can be problematic for things like decentralized markets.

Protecting against this is difficult and you will likely need to devise contract specific solutions.

Decentralized markets can mitigate concerns by implementing batch auctions or using a pre-commit scheme, where the details are submitted after the transaction is committed.

Integer Overflow and Underflow

Integers can underflow or overflow in the EVM.

The max value for an unsigned integer is 2 ^ 256 - 1, which is roughly 1.15 times 10 ^ 77. If an integer overflows, the value will go back to 0. For example, a variable called score of type uint8 storing a value of 255 that is incremented by 1 will now be storing the value 0.

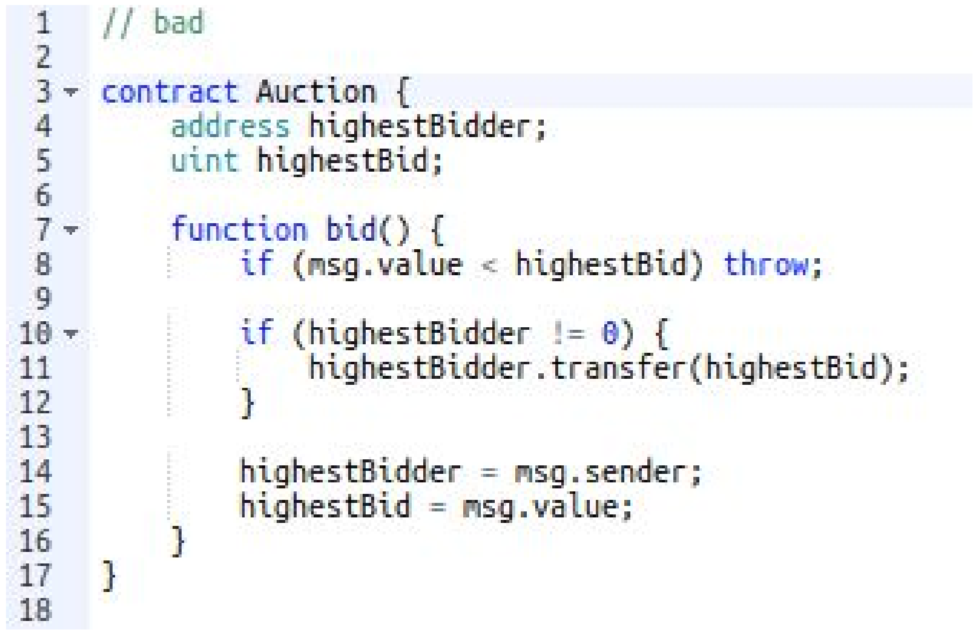
You may or may not have to worry about integer overflow depending on your smart contract.

A variable that can be set by user input may need to check against overflow, whereas it is infeasible that a variable that is incremented will ever approach this max value.

Underflow is a similar situation, but when a uint goes below its minimum value it will be set to its maximum value.

Be careful with smaller data types like uint8, uint16, etc… they can more easily reach their maximum value

Denial of Service



```
// bad

contract Auction {
    address highestBidder;
    uint highestBid;

    function bid() {
        if (msg.value < highestBid) throw;

        if (highestBidder != 0) {
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}

```

Another danger of passing execution to another contract is a denial of service attack.

In the provided example, the **highestBidder** could be another contract and transferring funds to the contract triggers the contract’s fallback function. If the contract’s fallback always reverts, the Auction contract’s bid function becomes unusable - it will always revert. The bid function requires the transfer operation to succeed to fully execute.

The contract at the provided address throws an exception, execution halts and the exception is passed into the calling contract and prevents further execution.

This problem is avoidable using the withdrawal pattern.

Force Sending Ether

Another danger is using logic that depends on the contract balance.

Be aware that it is possible to send ether to a contract without triggering its fallback function.

Using the selfdestruct function on another contract and using the target contract as the recipient will force the destroyed contract’s funds to be sent to the target.

It is also possible to precompute a contracts address and send ether to the address before the contract is deployed.

The contract’s balance will be greater than 0 when it is finally deployed.

Useful links:

A survey of attacks on Ethereum smart contracts

Solidity docs - Known bugs

A hacker stole $31M of Ether - how it happened, and what it means for Ethereum

This document is the adaptation of the presentation slides into a more easily readable format for reading.

Formal Verification

There is a dire need for more secure methods of smart contract development within the blockchain community.

There have been hundreds of millions of dollars worth of crypto assets lost or stolen due to insecure smart contracts, and there are thousands of smart contracts on the blockchain with known vulnerabilities.

Part of the problem is due to the current state of developer tools and programming languages such as Solidity.

The goal is to make writing secure smart contracts as easy and accessible as possible. One route to this goal is via formal verification.

From Wikipedia, **formal verification** is the act of proving or disproving the correctness of intended algorithms underlying a system with respect to a certain formal specification or property, using formal methods of mathematics.

Said another way, it is a way to prove that a program is correct for all inputs. This can ensure that a hacker cannot modify the contract to an unintended state.

Using formal verification, smart contract proofs can be checked by machines. Verification requires specific programming languages and features that are not currently present in the Ethereum ecosystem.

Read more about formal verification as it applies to computer science more generally on Wikipedia.

What is formal verification and why is it important for smart contracts?

Read about ongoing work in Formal Verification of Ethereum Contracts.

Useful links:

What is formal verification, and why is it important for smart contracts? - stackexchange

Formal Verification of Ethereum Contracts

Automated Formal verification tool