Lecture 20 - Smart Contract Pitfalls

Smart contracts have a unique set of limitings and pitfalls.

First put them in context. An analogy… Cryptocurency is to government-money, what smart-contracts are to law.

General Limitations on Solidity

1. no way to iterate over maps
2. data stored in arrays has to be returned 1 at a time
3. friction between 256bit ints and JavaScript
4. multiple-inheritance object oriented is just generally bad
5. solidity keeps changing
6. weird data types byres32 is same storage as int256/uint256, strings are 1 byte in a uint256 etc.
7. “address” type may or may not be “payable”
8. lots of “type-casting” required.

Limitations to the code.

A smart contract is a piece of code that is tied to a blockchain, triggered by transactions and data passed in the transactions and which reads and writes data in that blockchain’s history. The code behaves as if it is an embed software system. To maintain consistency between nodes in the chain - all nodes must see the same data. This means that the smart contract can not go pull the price of a stock itself - if node1 sees a different value than node2 they will get different output results. The data must be fixed and written to the chain before the smart contact is called, or it must be a parameter in the transaction and shared across all the nodes. This means that everything that takes place on a blockchain must be deterministic.

Smart contracts are forever.

Defects are forever. You can’t go back and change the contact and get a consistent new state for the block. Microsoft did a book called “Code Complete” that found that shipping code from Microsoft had 15-20 defects per 1000 lines of code. Some systems like NASA’s control system for the Shuttle had much fewer defects, close to 0 per 1000 lines of code. The cost for developing systems like that was 100,000x as much. So… For smart contracts which development system are you using?

Available forever.

Contracts don’t just go away. Version 1 of your contract is still there when you really want to be on version 3. If you want to stop people from using version one you have to have built into the contract a flag to say stop running. You have to protect the flag so that only the contract owner or some authorized list of people can set/unset the flag.

Upgrade-ability.

Since the contract and it’s address are forever - you have to plan a way to upgrade the contract. There is a “Proxy” system that has been used for this but upgrades could also be handled by forcing users to use a new-address and a new contract.

The “Proxy” model for upgrades is really nasty and complex.

```
// SPDX-License-Identifier: MIT
pragma solidity >=0.6.0 <=0.9.0;

import "@openzeppelin/contracts/proxy/Proxy.sol";

/**
 * @title Proxy
 * @dev Gives the possibility to delegate any call to a foreign implementation.
 */
contract ProxyToXXX is Proxy {

        address private owner;
        address private GrouceCreditImpl;

        uint256 public version;

        modifier onlyOwner() {
                require( msg.sender == owner, "Sender not authorized.");
                // Do not forget the "_;"! It will be replaced by the actual function
                // body when the modifier is used.
                _;
        }

    constructor( address _addr, uint256 _version ) Proxy() {
                GrouceCreditImpl = _addr;
                owner = msg.sender;
                version = _version;
        }

        /**
    * @dev Make `_newOwner` the new owner of this contract.
        */
        function changeOwner(address _newOwner) public onlyOwner() {
```

```
        function changeOwner(address _newOwner) public onlyOwner() {
                owner = _newOwner;
        }

        /**
    * @dev Upgrade the underling contract to a new version. `_newAddr` is the address o
        * the new contract. `_newVersion` is the new version number.
        */
        function upgradeContract(address _newAddr, uint256 _newVersion) public onlyOwne
                GrouceCreditImpl = _newAddr;
                version = _newVersion;
        }

        /**
    * @dev Tells the address of the implementation where every call will be delegated.
    * @return address of the implementation to which it will be delegated
    */
        function _implementation() internal view override  returns (address) {
                return ( GrouceCreditImpl );
        }

}
```

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

pragma solidity ^0.8.0;

/**
 * @dev This abstract contract provides a fallback function that delegates
 * all calls to another contract using the EVM instruction `delegatecall`. We
 * refer to the second contract as the _implementation_ behind the proxy,
 * and it has to be specified by overriding the virtual {_implementation}
 * function.
 *
 * Additionally, delegation to the implementation can be triggered manually
 * through the {_fallback} function, or to a different contract through
 * the {_delegate} function.
 *
 * The success and return data of the delegated call will be returned back
 * to the caller of the proxy.
 */
abstract contract Proxy {
  /**
   * @dev Delegates the current call to `implementation`.
   *
   * This function does not return to its internall call site, it will
   * return directly to the external caller.
   */
```

```
    function _delegate(address implementation) internal virtual {
      // solhint-disable-next-line no-inline-assembly
      assembly {
        // Copy msg.data. We take full control of memory in this inline
            // assembly block because it will not return to Solidity code.
            // We overwrite the Solidity scratch pad at memory position 0.
        calldatacopy(0, 0, calldatasize())

        // Call the implementation.
        // out and outsize are 0 because we don't know the size yet.
        let result := delegatecall(gas(), implementation, 0,
                          calldatasize(), 0, 0)

        // Copy the returned data.
        returndatacopy(0, 0, returndatasize())

        switch result
        // delegatecall returns 0 on error.
        case 0 { revert(0, returndatasize()) }
        default { return(0, returndatasize()) }
      }
    }

    /**
     * @dev This is a virtual function that should be overriden
     * so it returns the address to which the fallback function
     * and {_fallback} should delegate.
     */
    function _implementation() internal view virtual returns (address);

    /**
     * @dev Delegates the current call to the address
     * returned by `_implementation()`.
     *
     * This function does not return to its internall call
     * site, it will return directly to the external caller.
     */
    function _fallback() internal virtual {
      _beforeFallback();
      _delegate(_implementation());
    }

    /**
     * @dev Fallback function that delegates calls to the address
     * returned by `_implementation()`. Will run if no other
     * function in the contract matches the call data.
     */
    fallback () external payable virtual {
      _fallback();
    }
    /**
     * @dev Fallback function that delegates calls to the address
     * returned by `_implementation()`. Will run if call data
     * is empty.
     */
    receive () external payable virtual {
      _fallback();
    }

    /**
     * @dev Hook that is called before falling back to the
     * implementation. Can happen as part of a manual
     * `_fallback` call, or as part of the Solidity
     * `fallback` or `receive` functions.
     *
     * If overriden should call `super._beforeFallback()`.
     */
    function _beforeFallback() internal virtual {
    }
  }
```

Security is up to the contract writer. An example

```
constructor() public {
    initializeContract();
}
function initializeContract() public {
    owner = msg.sender;
}
```

You have to plan for the future - hard to do.

What happens when the author of the contract dies? What happens if the business goes under?

Integer underflow has to be considered in ALL calculations!

```
// Improper usage of integers
function withdraw(uint _amount) {
    require(balances[msg.sender] - _amount > 0);
```

```
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
    }


    // One corrct way
    function withdraw(uint _amount) {
        require(balances[msg.sender] >= _amount);
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
    }

    // Anotehr
    function withdraw(uint _amount) {
        require(_amount >= 0 && balances[msg.sender] >= 0 &&
                    balances[msg.sender] >= _amount);
        msg.sender.transfer(_amount);
        balances[msg.sender] -= _amount;
    }
```

Contract to contact calls must be carefully handled. (Or not used at all).

What happens if the “call” in the contract leads back to the same withdraw call?

```
    function withdraw(uint _amount) {
        require(balances[msg.sender] >= _amount);
        msg.sender.call.value(_amount)();                   // bad
        balances[msg.sender] -= _amount;
    }
```

You need other “things” in the contract that are usually not obvious.

```
    // SPDX-License-Identifier: MIT
    pragma solidity >=0.6.0 <=0.9.0;

    contract InsLogEvent {
            address payable owner;

            event AnEvent ( address indexed account, string msg );

            constructor() {

                    owner = payable(msg.sender);
            }

            modifier onlyOwner() {
                    require( msg.sender == owner, "Sender not authorized.");
                    // Do not forget the "_;"! It will be replaced by the actual function
                    // body when the modifier is used.
                    _;
            }

            // Make `_newOwner` the new owner of this contract.
            function changeOwner(address payable _newOwner) public onlyOwner() {
                    owner = _newOwner;
            }

            function IndexedEvent ( address _acct, string memory _msg ) public returns ( bo
                    // TODO - emit event
                    emit AnEvent ( _acct, _msg );
                    return (true);
            }

            // This function is called for all messages sent to
            // this contract, except plain Ether transfers
            // (there is no other function except the receive function).
            // Any call with non-empty calldata to this contract will execute
            // the fallback function (even if Ether is sent along with the call).
            fallback() external payable {}

            // This function is called for plain Ether transfers, i.e.
            // for every call with empty calldata.
            receive() external payable {}

            function withdraw( uint256 _amount ) public onlyOwner() {
                    owner.transfer(_amount);
            }

            // destroy the contract and reclaim the leftover funds.
        function kill() public onlyOwner() {
                    //      Calling selfdestruct(address) sends all of the contract's curre
            selfdestruct(payable(msg.sender));
        }
    }
```