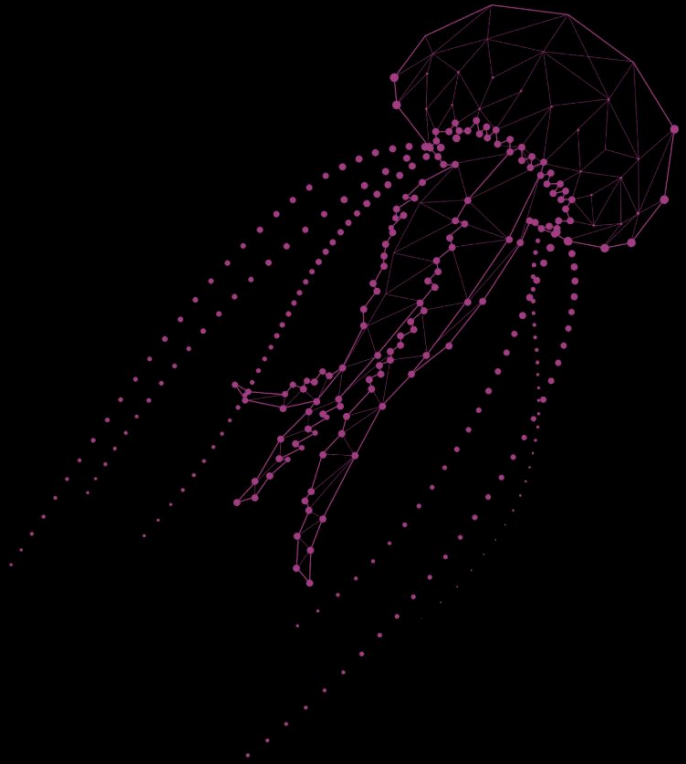
Ethereum Smart Contract Upgradability Workshop



TIEC - Cairo
April 14th,2019

Ahmed Ali <aabdulwahed>

ocean

Agenda

- Motivation
- Limitations
- Contract Structure
- EVM Data Locations
- Low Level Calls
- Upgradability Patterns
- Recommendations
- Tools: ZeppelinOS
- Discussion/networking

ocean

Motivation

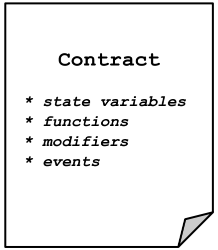
- Smart contracts are immutable code
- Any code is prone to errors and vulnerabilities
- Upgradability means the ability to upgrade smart contracts after they have been deployed!

Limitations

- Any old deployed contract will stay as is forever on Ethereum
- Users might not know about the release of a new contract version

Contract Structure

- **State variables:**
  - Permanently stored variables in contract storage.
  - A special type of state variables is *constant* state variable.
- **Events:**
  - EVM logging facility
  - Stored in the transaction log
  - Search optimization (indexed)
  - Support anonymous events

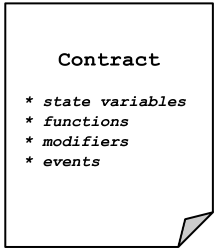


Ethereum Smart Contract Upgradability

Contract Structure

- **Functions:**
  - Executable units of code
  - Visibility (External, Public, internal, private)
  - constructor a special function, executed only once upon the contract creation

- **Modifiers:**
  - Modify the function behavior by automatically check a condition prior to function execution



Ethereum Smart Contract Upgradability

Contract Structure

```
contract SampleContract is Ownable {
    // state variables
    uint256 public value = 0;
    // modifier
    modifier onlyValidValue(uint256 _value)
    {
        require(
            value > 0,
            'Invalid value'
        );
        _;
    }
    // event
    event ValueUpdated(
        uint256 indexed oldValue,
        uint256 indexed newValue,
        uint256 updatedAt
    );
```

setValue() will never work and always revert with 'Invalid value`

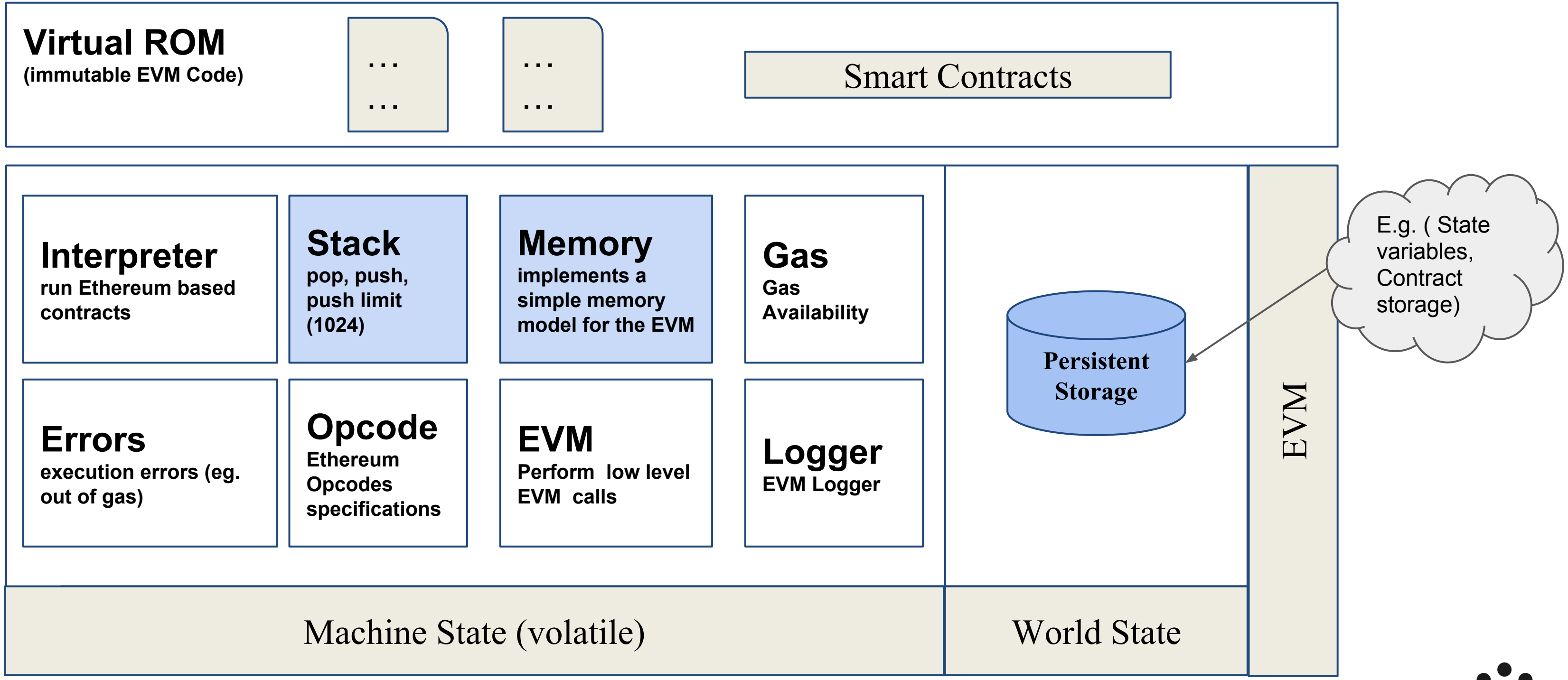
```
// constructor
constructor(address contractOwner)
    public
{
    require(contractOwner != address(0));
    transferOwnership(contractOwner);
}

function setValue(uint256 _value)
    public      // visibility is public
    onlyOwner() // inherited modifier from Ownable!
    returns(bool)
{
    return _setValue(_value);
}

function _setValue(uint256 newValue)
    private
    onlyValidValue(newValue)
    returns(bool)
{
    uint256 oldValue = value;
    value = newValue;
    emit ValueUpdated(oldValue, newValue, block.number);
}
```

https://github.com/aabdulwahed/contract-upgradability-Cairo-workshop-2019/tree/master/labs/sample_contract

ocean

Ethereum Smart Contract Upgradability

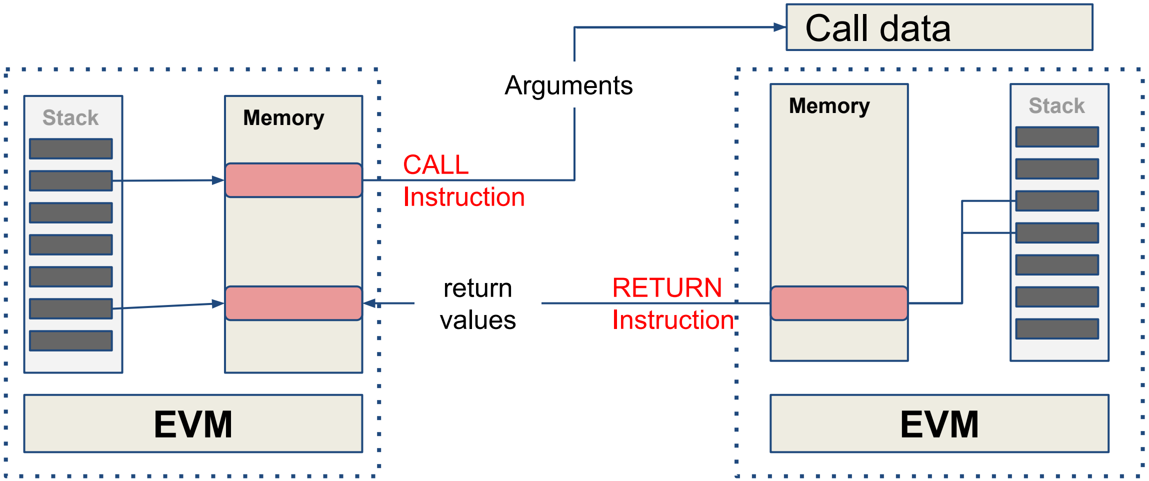
Data Management



EVM Data Locations

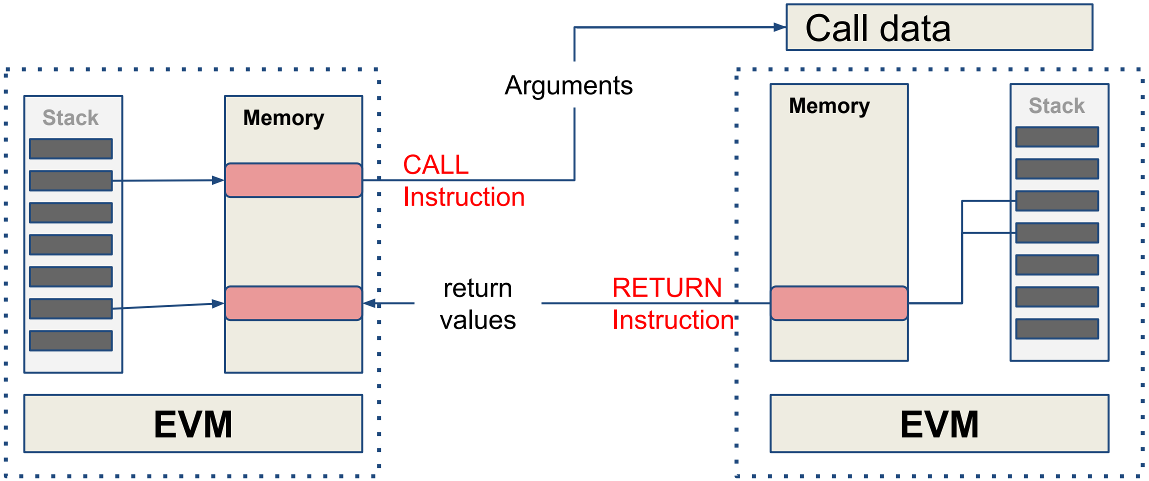
- **Storage:**
  - Persistent read-write word-addressable space
  - Storage is a key-value store that maps 256-bit words to 256-bit words.
  - Access with SSTORE/SLOAD instructions
  - All locations in storage are well-defined initially as zero
  - *SLOAD* loads a word from storage into the stack
  - *SSTORE* saves a word to storage
- **How does storage allocation work?**
  - Statically sized variables
    - They are laid out contiguously in storage starting from 0
  - Mapping and Dynamically-sized Arrays
    - Starts at unfilled slot in the storage at some position *P*
    - For arrays, the data is stored in *keccak256(p)*
    - For mappings, for key at position P, the value is stored in keccak256(k . p)

EVM Data Locations



- **Stack:**
  - EVM is a 256-bit word machine
  - The stack has a maximum size of 1024
  - All EVM operations are performed on the stack
  - Operations are represented by opcodes (POP, PUSH, DUP, SWAP, ect.)
  - The depth of transaction invocation (message call) is limited to less than 1024 levels

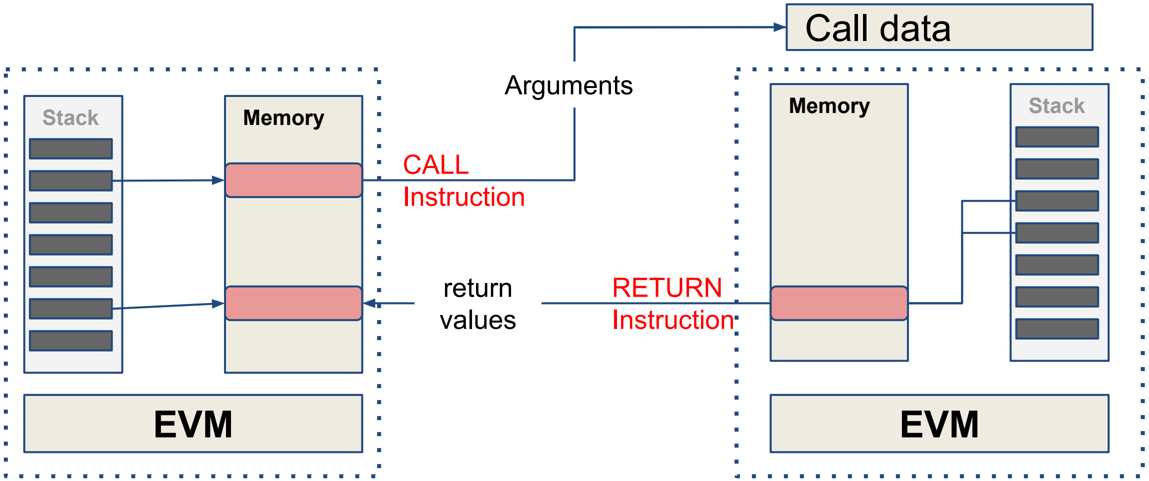
EVM Data Locations



- **`Memory`:**
  - `Read-write byte-addressable space (word size: 256 bits)`
  - `Memory is linear and can be addressed at byte level`
  - `Access with MSTORE/MSTORE8/MLOAD instructions`
  - `All locations in memory are predefined as zero`

Ethereum Smart Contract Upgradability

EVM Data Locations



- **Calldata:**
  - Read-only byte-addressable space
  - Holds data parameters of a transaction or call
  - Unlike stack, in order to read this data, you have to specify byte offset and number of bytes

EVM Data Locations

- **Play with Data locations (inline assembly example)**

```
contract PlayWithInLineAssembly {
function add(uint256 _a, uint256 _b) public pure
  returns (uint256 result)
  {
    assembly {
      // Solidity always stores a free memory pointer at position 0x40
      // load into stack from memory @0x40
      let aPtr := mload(0x40)
      // increment bPtr by adding 32 bytes offset to 0x40
      let bPtr := add(aPtr, 32)
      // copy call data (_a) into memory: after first 4 bytes (function selector)
      calldatacopy(aPtr, 4, 32)
      // copy call data (_b) into memory: after first (4bytes + 32 bytes)
      calldatacopy(bPtr, add(4, 32), 32)
      // load data (aPtr, bPtr) from memory into stack
      result := add(mload(aPtr), mload(bPtr)) // sum two data values and assign the output to result
    }
  }
}
```

https://github.com/aabdulwahed/contract-upgradability-Cairo-workshop-2019/tree/master/labs/data_locations

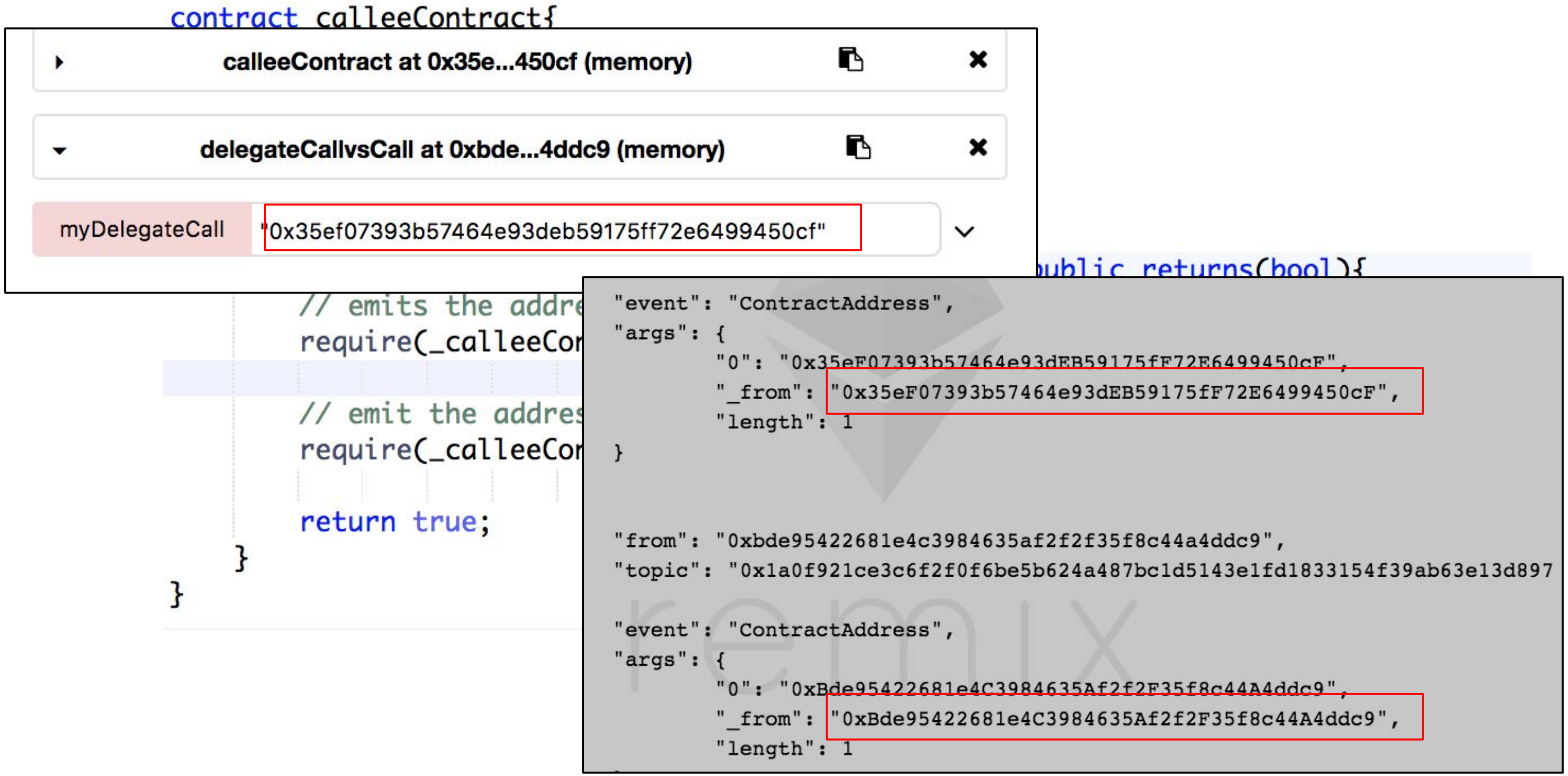
Low Level Calls

Special Low level functions in solidity: *call vs Delegatecall*

```
contract calleeContract{
    event ContractAddress(address _from);
    function triggerCall() public payable {
        emit ContractAddress(this);
    }
}
contract delegateCallvsCall {
    function myDelegateCall(address _calleeContract) public returns(bool){
        // emits the address of the callee contract
        require(_calleeContract.call(bytes4(keccak256('triggerCall()'))),
                                     'invalid low level call');
        // emit the address of this contract address (caller)
        require(_calleeContract.delegatecall(bytes4(keccak256('triggerCall()'))),
                                     'invalid low level delegatecall');
        return true;
    }
}
```

Low Level Calls

Special Low level functions in solidity: *call vs Delegatecall*



Upgradability Patterns

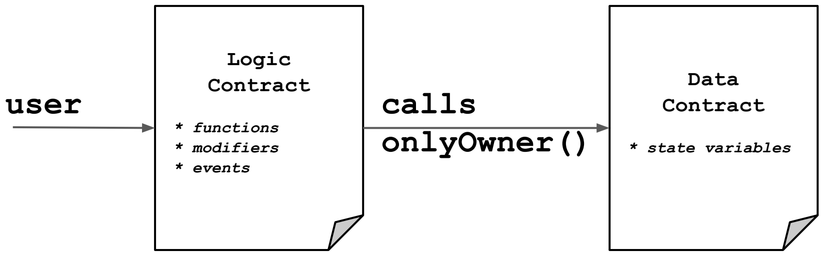
Two families of patterns

Data Separation	**Delegatecall-based proxies**
This family relies on separation of logic and data. The logic contract is the only authorized contract that can call the data contract	Data and logic contracts are kept separate but the proxy contract (data contract) calls the logic contract via *delegatecall*

Upgradability Patterns

Data Separation

- The design is simple, does not require any low level expertise
- Only the owner can alter its content
- For upgradability, we need to understand **how to store data,** and **how to perform the upgrade.**



ocean

Upgradability Patterns

Data Separation

- **How to store data?**
  - Eternal Storage pattern
  - Unified key-value data storage pattern
  - A mapping from a *bytes32* key value to each base variable type

Mappings	
Key	**Value**
bytes32	uint256
bytes32	int8
bytes32	string

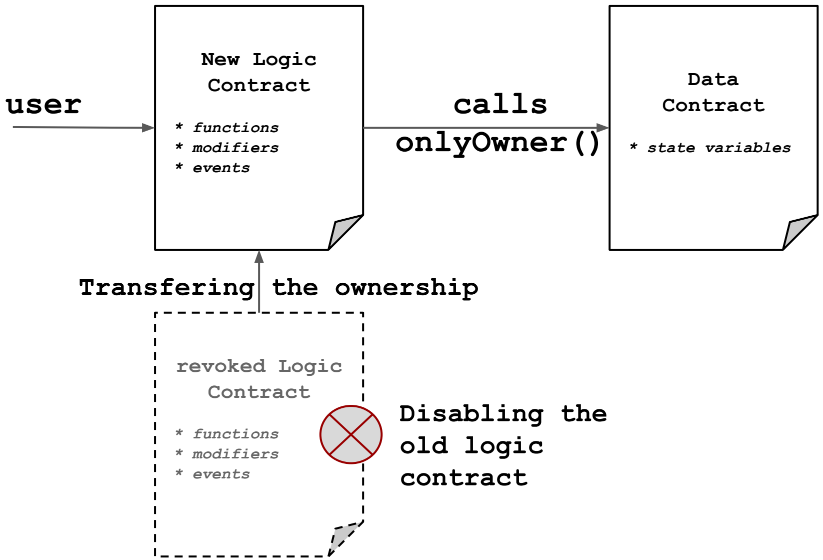
```
contract EternalStorageDataContract is Ownable {

    mapping(bytes32 => uint256) uInt256Storage;

    constructor(address contractOwner)
        public
    {
        require(contractOwner != address(0));
        transferOwnership(contractOwner);
    }

    function getUint256(bytes32 key)
        public
        view
        returns(uint256)
    {
        return uInt256Storage[key];
    }

    function setUint256(bytes32 key, uint new_val)
        public
        onlyOwner
    {
        uInt256Storage[key] = new_val;
    }
}
```

https://github.com/aabdulwahed/contract-upgradability-Cairo-workshop-2019/tree/master/labs/eternal_storage

ocean

Data Separation

- **How to perform upgrade?**
  - Mechanisms
    - Puasable mechanism by deploying a new logic contract and transferring the ownership of the data contract to it

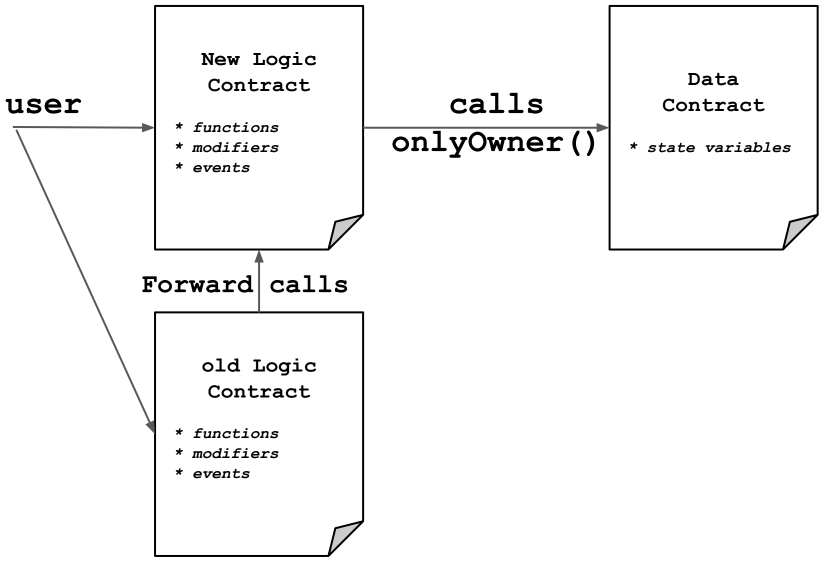


Pausable Mechanism

Upgradability Patterns

Data Separation

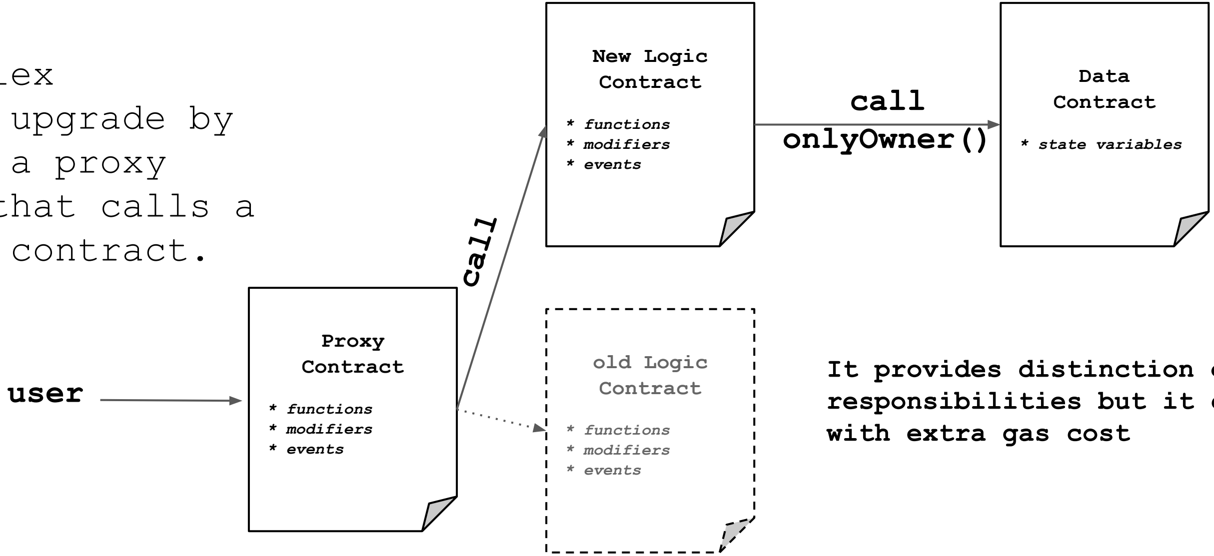
- **How to perform upgrade?**
  - Mechanisms
    - Different approach by forwarding the calls from the original logic contract to the new version



Forwarding Mechanism

Data Separation

- **How to perform upgrade?**
  - Mechanisms
    - More complex approach, upgrade by deploying a proxy contract that calls a new logic contract.



It provides distinction of responsibilities but it comes with extra gas cost

Proxy Mechanism

ocean

Upgradability Patterns

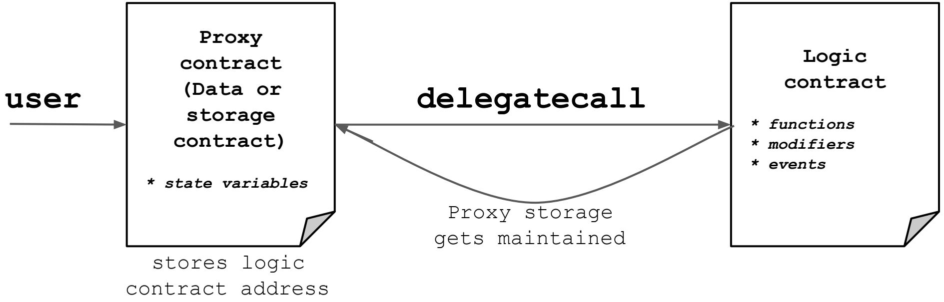
Data Separation

- **Risks**
    - Adds more complex authorization schema to the code
    - Eternal storage increases the complexity of the data model.
    - Some developers might implement this pattern incorrectly (e.g keeping some logic in data contract which is impossible to upgrade)

Upgradability Patterns

Delegatecall-based proxies

- **Similarly to Data Separation approach**
  - This approach splits the contract into two contracts:
    - Proxy contract which holds data
    - Logic contract holds logic
  - But the proxy contract (data contract) calls the logic contract using ***delegatecall*** in the context of the proxy contract



Upgradability Patterns

Delegatecall-based proxies

Bad delegatecall usage
(Expect memory corruption)

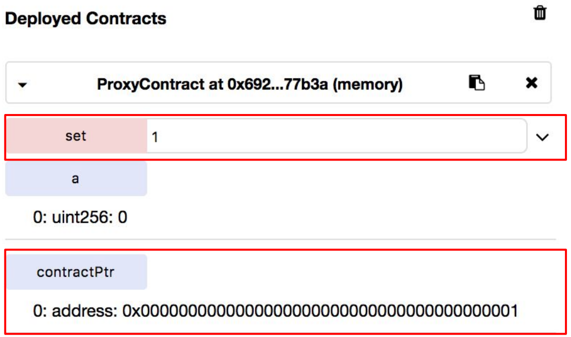
```
contract SampleLogicContract {
    uint public a;

    function set(uint val)
        public
        returns (bool)
    {
        a = val;
        return true;
    }
}
```

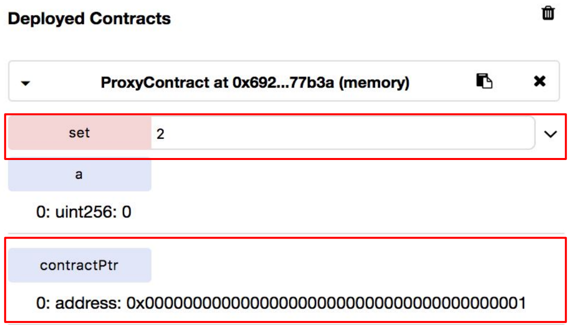
```
contract ProxyContract {
    address public contractPtr;
    // different state variable with the same name
    uint public a;

    constructor()
        public
    {
        contractPtr = address(new SampleLogicContract());
    }

    function set(uint val)
        public
        returns (bool)
    {
        bool state = contractPtr.delegatecall(bytes4(keccak256("set(uint256)")), val);
        require(
            state,
            'invalid delegatecall'
        );
        return true;
    }
}
```

https://github.com/aabdulwahed/contract-upgradability-Cairo-workshop-2019/tree/master/labs/bad_delegatecall_proxy

Delegatecall-based proxies

Bad delegatecall usage (Expect memory corruption)





- `SampleLogicContract.set` is executed within the context of `ProxyContract`
- `SampleLogicContract` knows only one state variable (a)
- If we try to execute set function in the context of ProxyContract, the delegate call will write the value to the first state variable **contractPtr** instead of **a**

https://solidity.readthedocs.io/en/v0.4.25/miscellaneous.html#layout-of-state-variables-in-storage

Upgradability Patterns

Delegatecall-based proxies

Safe delegatecall proxy

```
contract SampleLogicContract {
    uint public a;

    function set(uint val)
        public
        returns (bool)
    {
        a = val;
        return true;
    }
}
```

```
contract ProxyContract {
    uint public a;
    address public contractPtr;

    constructor()
        public
    {
        contractPtr = address(new SampleLogicContract());
    }

    function set(uint val)
        public
        returns (bool)
    {
        bool state = contractPtr.delegatecall(bytes4(keccak256("set(uint256)")), val);
        require(
            state,
            'invalid delegatecall'
        );
        return true;
    }
}
```

https://github.com/aabdulwahed/contract-upgradability-Cairo-workshop-2019/tree/master/labs/safe_delegatecall_proxy

`Delegatecall-based proxies`

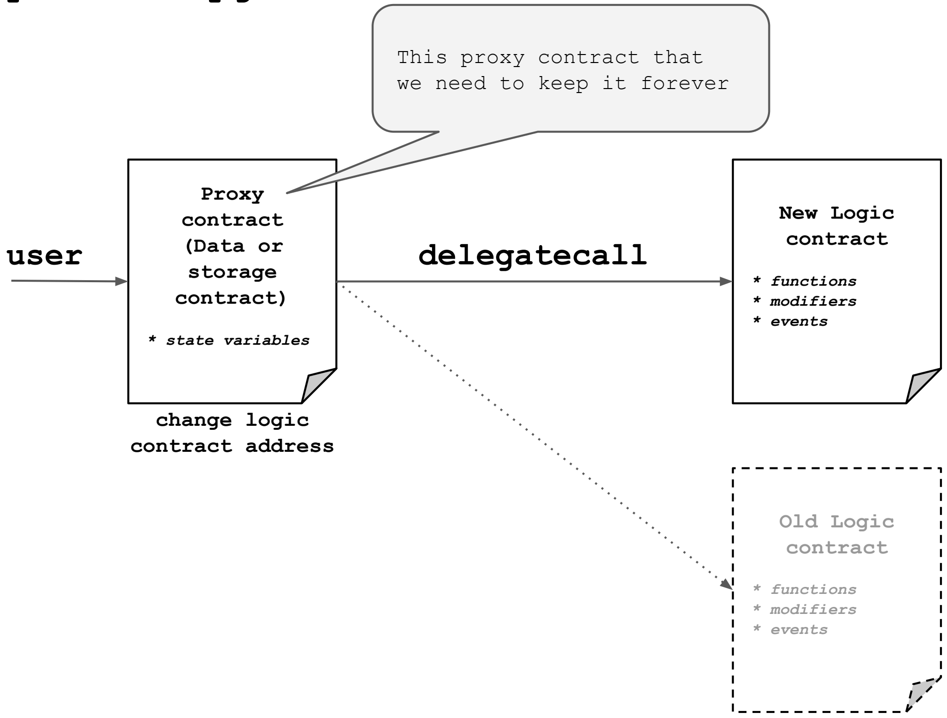
`Safe delegatecall proxy`

https://github.com/aabdulwahed/contract-upgradability-Cairo-workshop-2019/tree/master/labs/safe_delegatecall_proxy

Upgradability Patterns

Delegatecall-based proxies

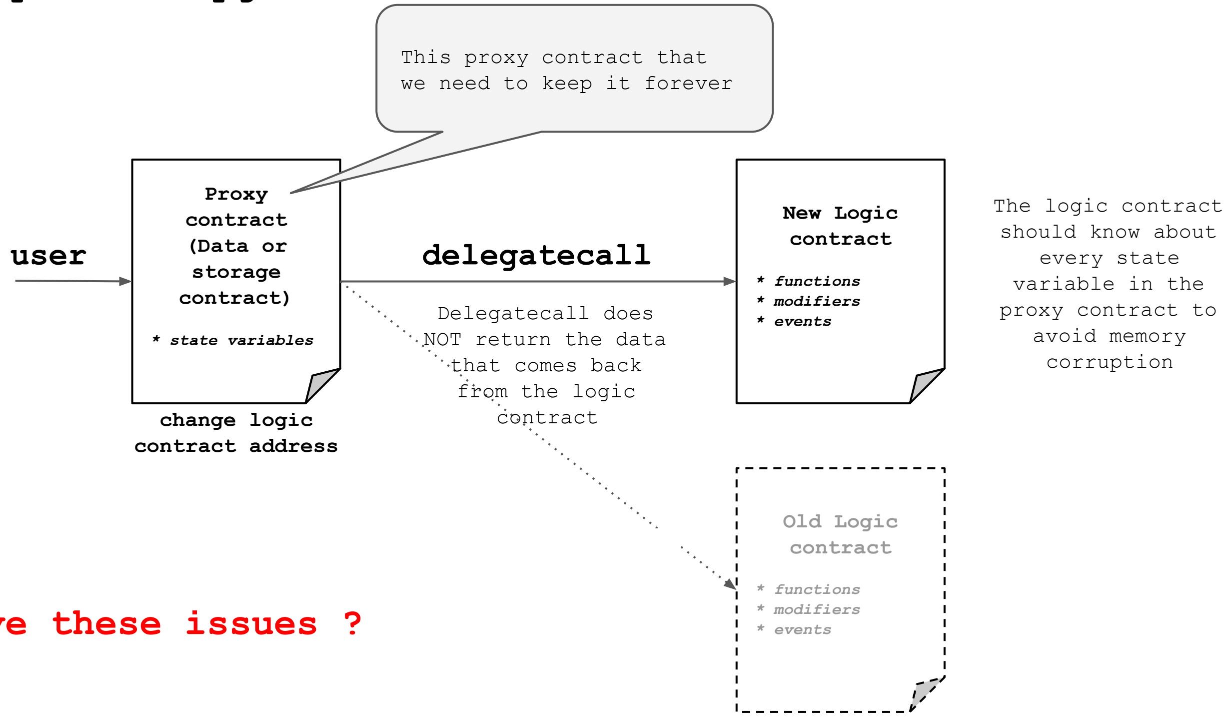
- **How to perform upgrade**



Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**



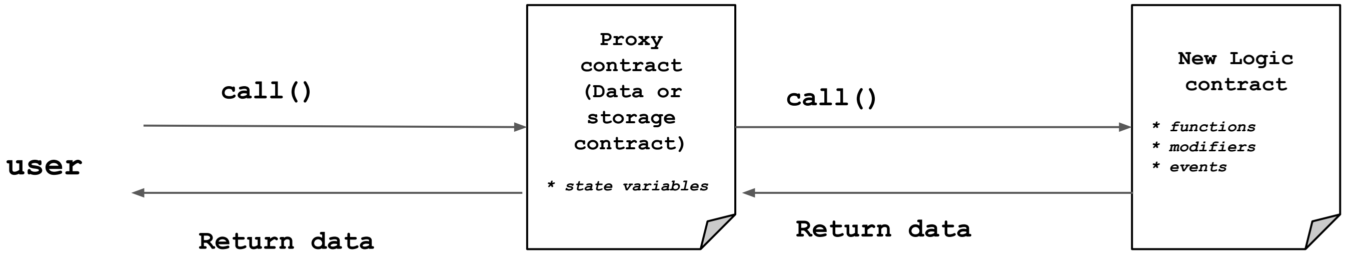
How to solve these issues ?

ocean

Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**



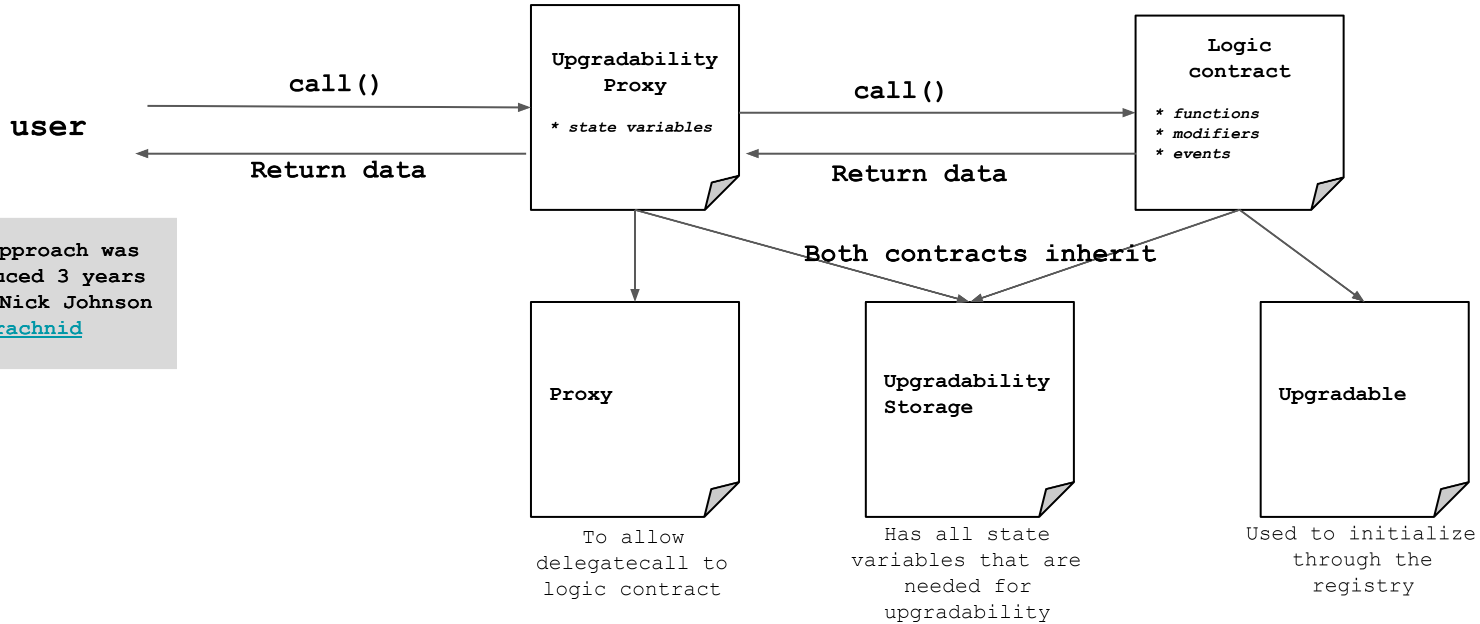
storage patterns		
Inherited Storage	Eternal Storage	Unstructured Storage

[ZeppelinOS Proxy Patterns]

Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**
  - **Inherited Storage** (A.k.a, simple Storage)



This approach was introduced 3 years ago by Nick Johnson [Arachnid]

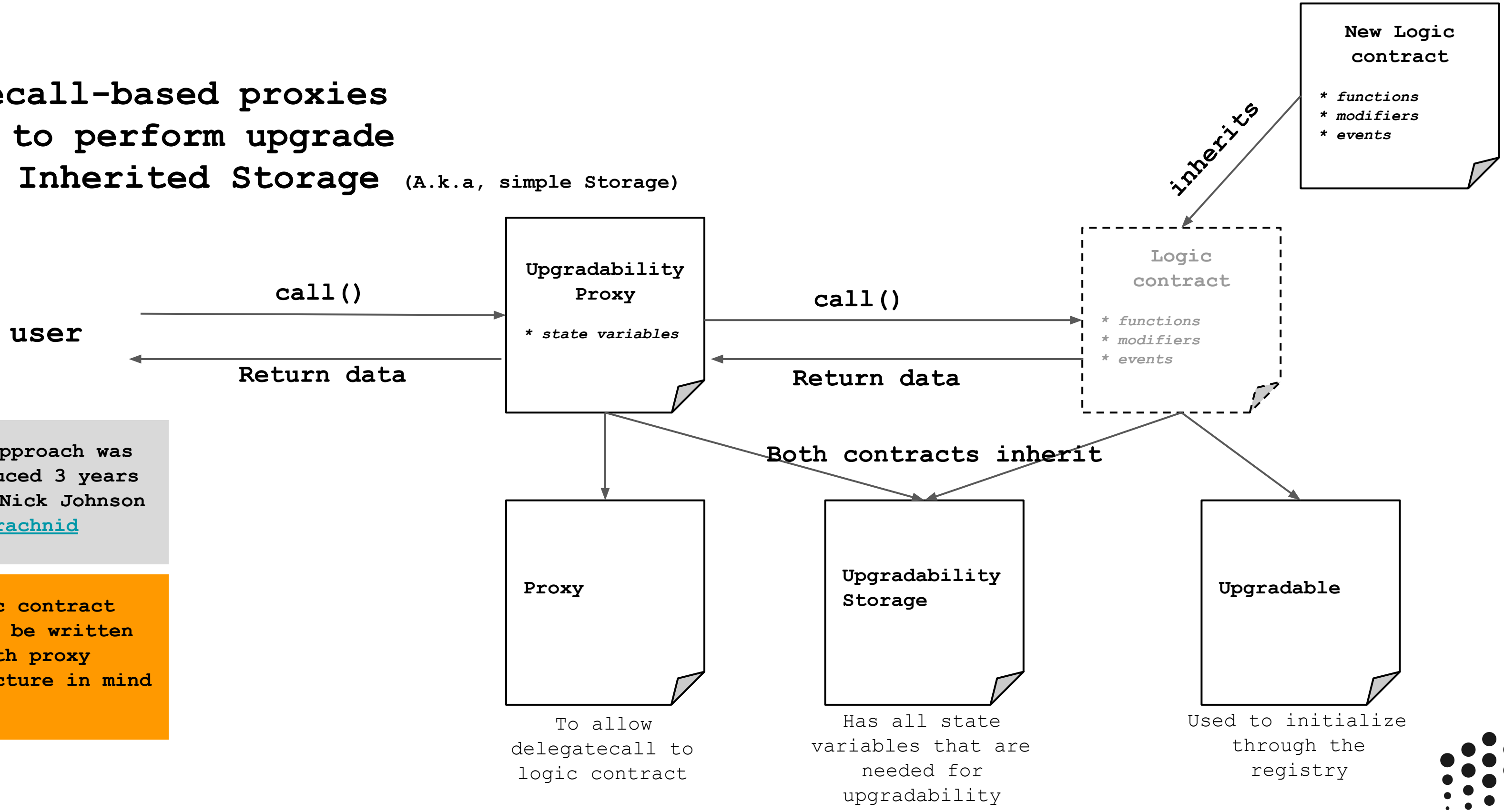
[ZeppelinOS Proxy Patterns]

[Upgradability using inherited storage]

Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**
  - **Inherited Storage** (A.k.a, simple Storage)



This approach was introduced 3 years ago by Nick Johnson Arachnid

Logic contract should be written with proxy architecture in mind

ZeppelinOS Proxy Patterns

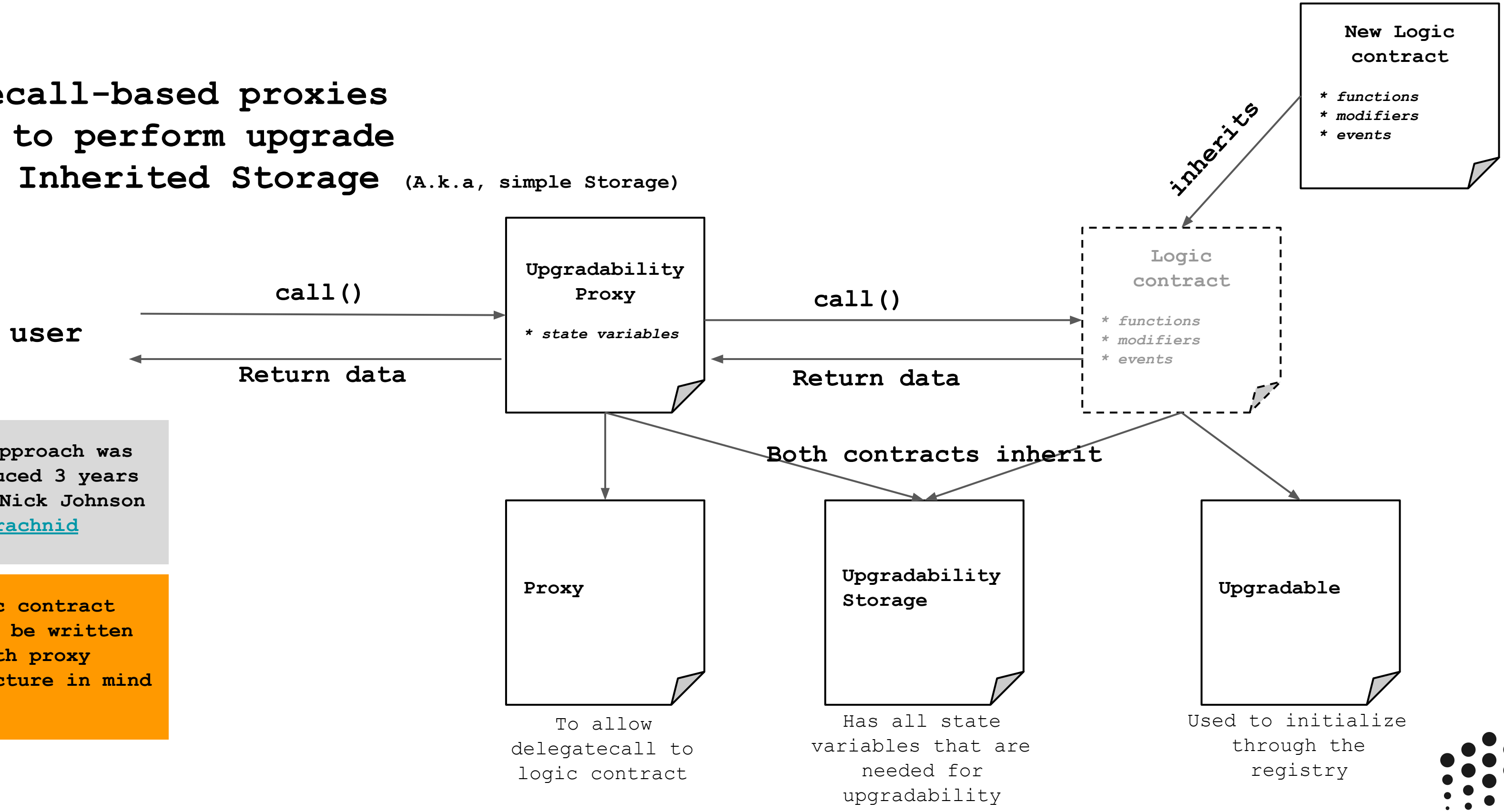
Upgradability using inherited storage

ocean

Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**
  - **Inherited Storage** (A.k.a, simple Storage)



This approach was introduced 3 years ago by Nick Johnson [Arachnid]

Logic contract should be written with proxy architecture in mind

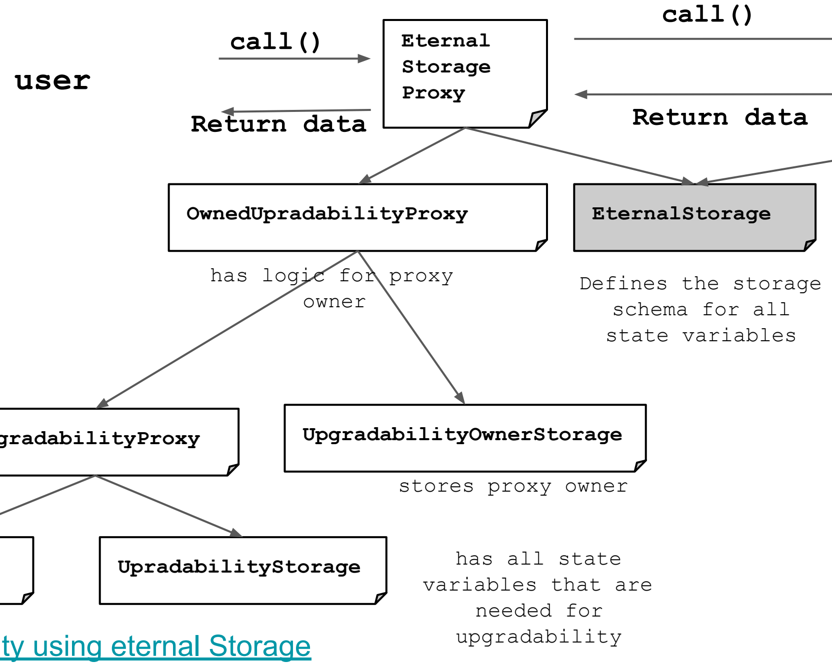
Can we have something better ? [Upgradability using inherited storage]

Ethereum Smart Contract Upgradability

Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**
  - **Eternal Storage**



to allow delegatecall to logic contract

[ZeppelinOS Proxy Patterns]

[Upgradability using eternal Storage]

Pros	Cons
The proxy is going be aware of all the state variables because all of them are defined in the same EternalStorage contract	*You have put all the state variables in EternalStorage. Future logic contracts should NOT define new state variables*

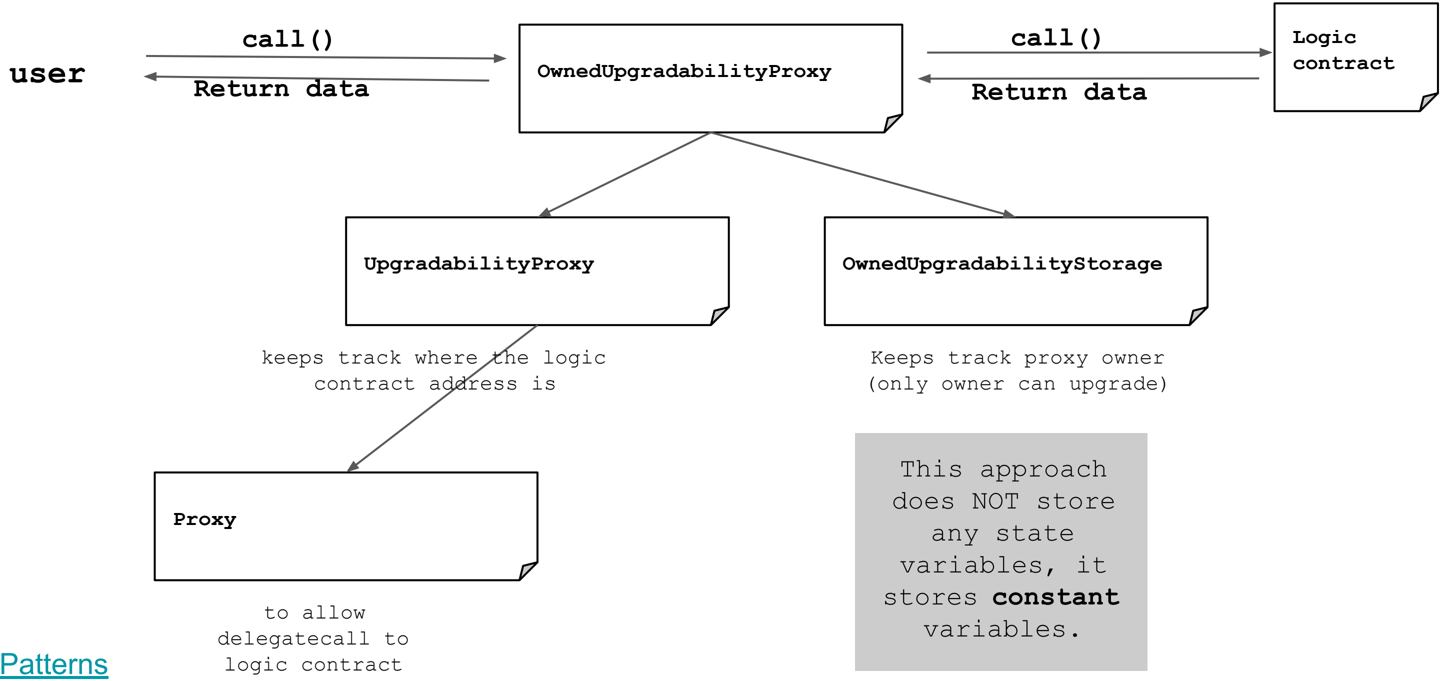
Logic contract should be written with proxy architecture in mind

ocean

Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**
  - Unstructured Storage



This approach does NOT store any state variables, it stores **constant** variables.

[ZeppelinOS Proxy Patterns]

ocean

Upgradability Patterns

Delegatecall-based proxies

- **How to perform upgrade**
  - **Unstructured Storage**
    - **Logic contract:** No additional requirements!

Cons
It might have a chance of hash collision for the constant position and another storage allocation. But it has a low probability

```
contract UpgradabilityProxy is Proxy {
    bytes32 private constant implementationPosition = keccak256("com.oceanprotocol.implementation");

    function implementation()
        public
        view
        returns(address implementationAddress)
    {
        bytes32 position = implementationPosition;
        assembly {
            implementationAddress := sload(position)
        }
    }

    function setImplementation(address newImplementationAddress)
        internal
    {
        bytes32 position = implementationPosition;
        assembly{
            sstore(position, newImplementationAddress)
        }
    }
}

contract OwnedUpgradabilityProxy is UpgradabilityProxy {

    bytes32 private constantpProxyOwnerPosition = keccak256("com.oceanprotocol.proxy.owner");
}
```

Pros
The proxy contract does not store any state variables, therefore the logic contract can be written without any proxy architecture

[ZeppelinOS Proxy Patterns]

Recommendations

- Have a detailed understanding of Ethereum internals
- Carefully consider the order of inheritance
- Carefully consider the order in which variables are declared
- Be aware that the compiler may use padding and/or pack variables together.
- Confirm that the variables' memory layout is respected
- Carefully consider the contract's initialization.
- Carefully consider names of functions in the proxy

ocean

Tools: ZeppelinOS

- **Key features:**
  - New platform with support for EVM packages(on-chain standard libraries)
  - Safer and cheaper upgradeability
    - initialization checks
    - storage layout checks
    - safe code checks
  - New governance systems
    - Using multi-signature wallet

https://blog.zeppelinos.org/zeppelinos-2-0/

Tools: ZeppelinOS

[ZeppelinOS Lab]

ocean

Tools: ZeppelinOS

ZeppelinOS Upgradability checklist

- ❏ You must add initializers.
- ❏ Don't forget to initialize the inherited contracts
- ❏ Make sure that all the initial values are set in an initializer function. Otherwise, any upgrable instance will not have these fields sets
- ❏ When you create new instance of an arbitrary contract inside contract, pass the instance of that contract to initialize function
- ❏ You can create new contract instances on the fly using BaseApp in ZeppelinOS
- ❏ Remember nothing prevent malicious actor from sending transactions

Tools: ZeppelinOS

ZeppelinOS Upgradability checklist

- ❑ Make sure to add the new variables at the end
- ❑ Be careful when you introduce new updates to the contracts
- ❑ Be careful with the solidity inheritance linearization
  - ❑ Avoid base contracts swapping
  - ❑ Avoid adding new variables to the base contracts.
  - ❑ A workaround for this is to declare unused variables on base contracts that you may want to extend in the future, as a means of "reserving" those slots.
  - ❑ Note that this trick does not involve increased gas usage.

Thank You!

ocean