Demystifying Proxies

Solidity Deep Dive

CodeMentor Jan 19 2023

Agenda

- Smart Contract Dev for Bulla Network 👨🏽‍💻
- Teach a Solidity Deep Dive Meetup 🤿
- Based in Denver 🏔️
- Github - Twitter - Streams

Talk Repo

Proxies

Agenda

- Required Knowledge
- Proxy Types:
  - Minimal Proxy (EIP 1167)
  - Transparent Proxy
  - UUPS
- Q/A + exploration

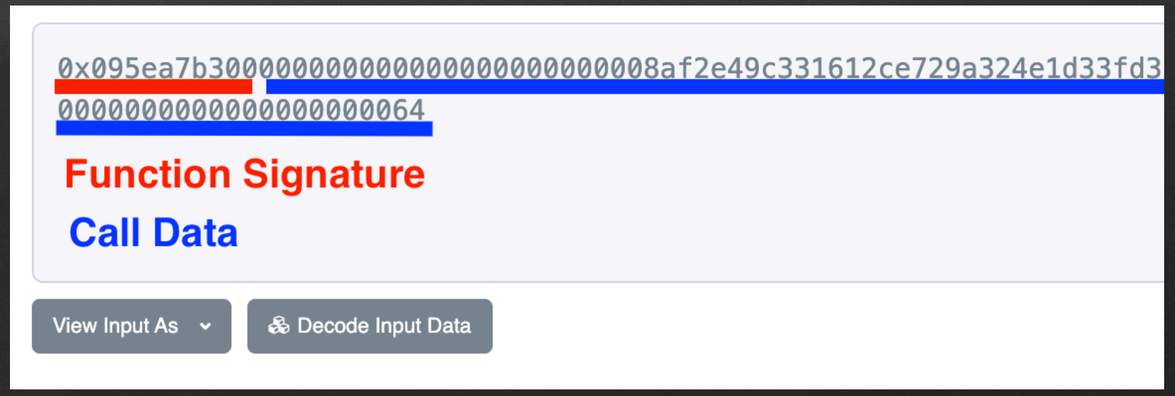
Prerequisites

- Transaction
- Delegate Call
- Storage Slots
- Fallback

Transactions

- Bytes 0-4 = function selector
- Bytes 4-n = calldata
- The function selector is the left-most 4 bytes of the keccak hash of the function signature: i.e keccak256(“totalSupply()”)
- <u>A normal TX:</u>
- <u>A TX with data:</u>

From	Fund sender, an EOA (20-byte address)
To	Fund recipient, another EOA (20-byte address)
Value	Amount, in weis
Data / Input	Empty
Gas Limit	Larger enough for an ether transfer transaction
Gas Price	To be determined by transaction initiator



Transactions (cont.)

- A TX will execute code on the receiver
- All calldata gets interpreted by the “Hub"
- Takes the first 4 bytes of the function and jumps the program to that function
- If the doesn’t match, I’ll trigger the *fallback* function

```
let selector := shift_right_224_unsigned(calldataload(0))
switch selector
case 0x06fdde03 {
    // name()
    external_fun_name_20()
}
case 0x095ea7b3 {
    // approve(address,uint256)
    external_fun_approve_104()
}
case 0x18160ddd {
    // totalSupply()
    external_fun_totalSupply_26()
}
```

Fallback

- You can specify a “fallback function” to catch any non-matching function selector
- If the first 4 bytes of call data doesn’t match any function specified in the hub, the fallback is called

```
contract CoolTokenWithFallback is ERC20 {
    constructor() ERC20("Cool token", "COOL", 18) {
        _mint(msg.sender, 10 ether);
    }

    fallback() external {
        _mint(msg.sender, 1 ether);
    }

    receive() external payable {
        _mint(msg.sender, 100 ether);
    }
}
```

Storage Slots

- A smart contract's storage is an array ranging from 0 -> 2^256-1
- The visual (top to bottom) position in the contract indicate the index based on solidity's storage packing rules 🧐
- Storage writes can be read as: "take value x, store it in slot y" 🤓

```
contract StorageSlots {
    uint256 variable1; // slot 0
    bytes32 variable2; // slot 1
    uint128 halfSizeVar; // slot 2
    uint128 halfSizeVar2; // slot 2
    bytes32 other; // slot 3

    function setVar1(uint256 newVar) public  {
        variable1 = newVar;
        // this tells the EVM to write newVar at storage slot 0
    }
```

Storage Slots (cont.)

- With assembly, you can write and read to any slot ✍️
- With assembly, you could completely DIY a contract's storage

```
contract StorageSlots {
    uint256 variable1; // slot 0
    bytes32 variable2; // slot 1

    function getVar2() public view returns (bytes32 _var2) {
        assembly {
            _var2 := sload(1) // args: [slot position to read]
        }
    }
    function setVar2(bytes32 newVar) public {
        assembly {
            sstore(1, newVar) // args [slot position to write, value]
        }
    }
}
```

```
uint256 constant MY_RANDO_SLOT = 124352352352362151212;

contract StorageSlots {
    function getVar1() public view returns (bytes32 _var2) {
        assembly {
            _var2 := sload(MY_RANDO_SLOT)
        }
    }
    function setVar1(uint256 newVar1) public {
        assembly {
            sstore(MY_RANDO_SLOT, newVar1)
        }
    }
}
```

Storage Slots (magic 🧙🏼)

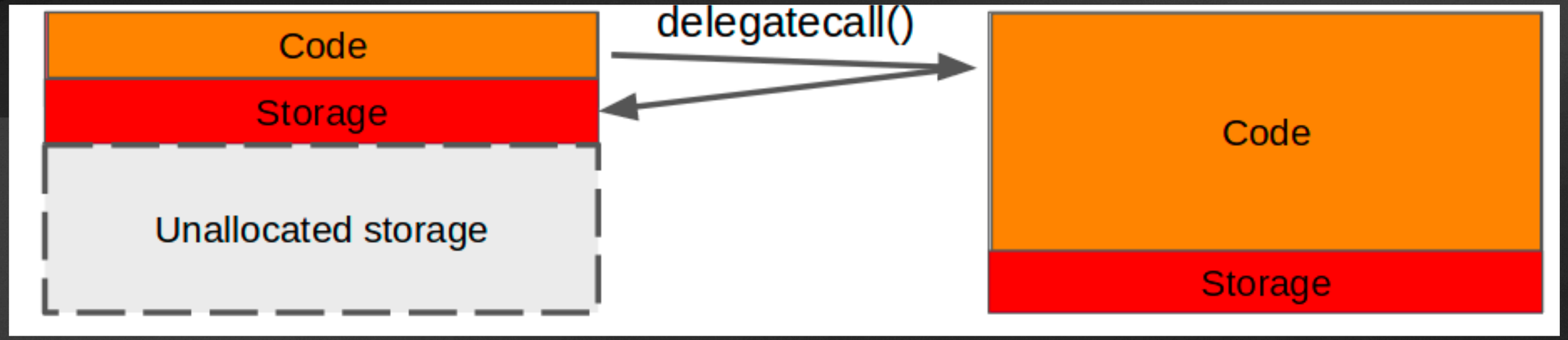
- Notice in packing rules, how structs follow the same packing rule as storage
- In assembly, **`slot`** is a member of any struct. It refers to the starting slot position of the struct in storage.
- Combining these techniques, you can create reusable storage read/writes to a custom slot in storage 🤯

```
contract StorageSlots {
    uint256 variable1; // slot 0
    bytes32 variable2; // slot 1
    uint128 halfSizeVar; // slot 2
    uint128 halfSizeVar2; // slot 2
}

struct StorageLayout {
    uint256 variable1; // slot 0
    bytes32 variable2; // slot 1
    uint128 halfSizeVar; // slot 2
    uint128 halfSizeVar2; // slot 2
}
```

```
uint256 constant MY_RANDO_SLOT = 124352352352362151212;
struct StorageLayout {
    uint256 var1;
}
contract StorageSlots {
    function getStorage() internal pure returns (StorageLayout storage) {
        StorageLayout storage customStorage;
        assembly {
            customStorage.slot := MY_RANDO_SLOT
        }
        return customStorage;
    }
}
```

Delegate call



Delegate call

- Allows for that contract to modify state
- msg.sender, msg.data, msg.value are the same

```
contract IndecisivePartner {
    string public dinnerLocation;

    function whereShouldWeGoToDinner(address partner) {
        (bool success, ) = partner.delegatecall(
            bytes4(keccak256("dinnerPlans()"))
        );
        if(!success) revert("couples therapy");
    }
}
```

```
contract AssertivePartner {
    string public dinnerLocation;

    function dinnerPlans() public {
        dinnerLocation = "Illegal Pete's    ;
    }                          🌮"
}
```

Delegate call

- Allows for that contract to modify state
- msg.sender, msg.data, msg.value are the same

```
contract IndecisivePartner {
    string public dinnerLocation;

    function whereShouldWeGoToDinner(address partner) {
        (bool success, ) = partner.delegatecall(
            bytes4(keccak256("dinnerPlans()"))
        );
        if(!success) revert("couples therapy");
    }
}
```

Delegates to Assertive Partner

```
contract AssertivePartner {
    string public dinnerLocation;

    function dinnerPlans() public {
        dinnerLocation = "Illegal Pete's    ;
    }                        🌮"
}
```

Delegate call

- Allows for that contract to modify state
- msg.sender, msg.data, msg.value are the same

```
contract IndecisivePartner {
    string public dinnerLocation;

    function whereShouldWeGoToDinner(address partner) {
        (bool success, ) = partner.delegatecall(
            bytes4(keccak256("dinnerPlans()"))
        );
        if(!success) revert("couples therapy");
    }
}
```

Modifies indecisive's storage

```
contract AssertivePartner {
    string public dinnerLocation;

    function dinnerPlans() public {
        dinnerLocation = "Illegal Pete's    ;
    }                          🌮"
}
```

Basic Proxy Recipe 👩‍🍳

A “logic contract” address

+

A fallback function

(which)

Forwards all calls to the logic contract

=

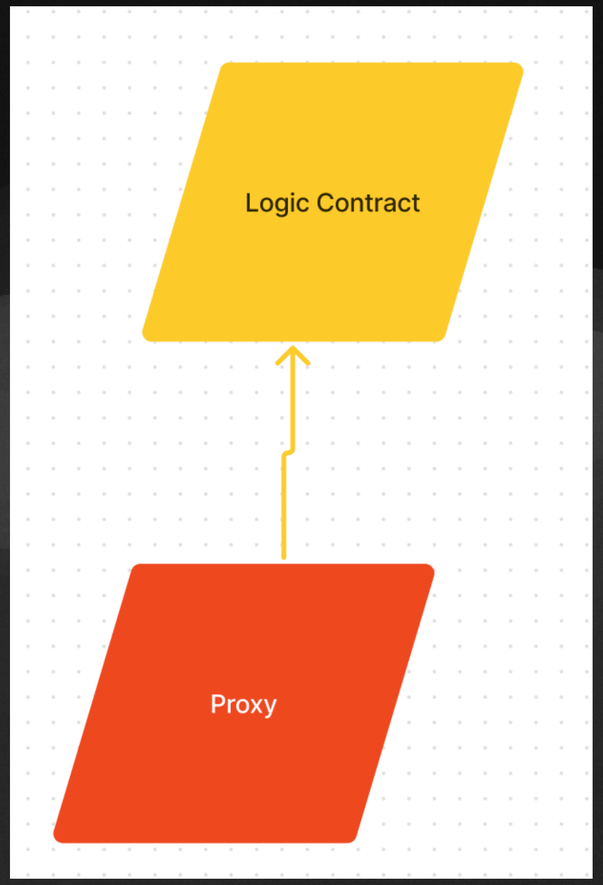
A contract where:

- Storage is kept at a single address
- Logic is determined by another address
- Logic is upgradable 💥

Proxy Types #1: Minimal Proxy

To create a proxy we need to:

1. Load the calldata
2. delegatecall the calldata to another contract
3. Copy the return data
4. Return the return data if it worked, revert if it failed



Proxy Types #1: Minimal Proxy

Bytecode is literally:
3d602d80600a3d3981f3363d3d373d3d3d363d73bebebebebebebebebebebebebebebebebebebebebebebebebebe5af43d82803e903d91602b57fd5bf3

- Super cheap 🔥

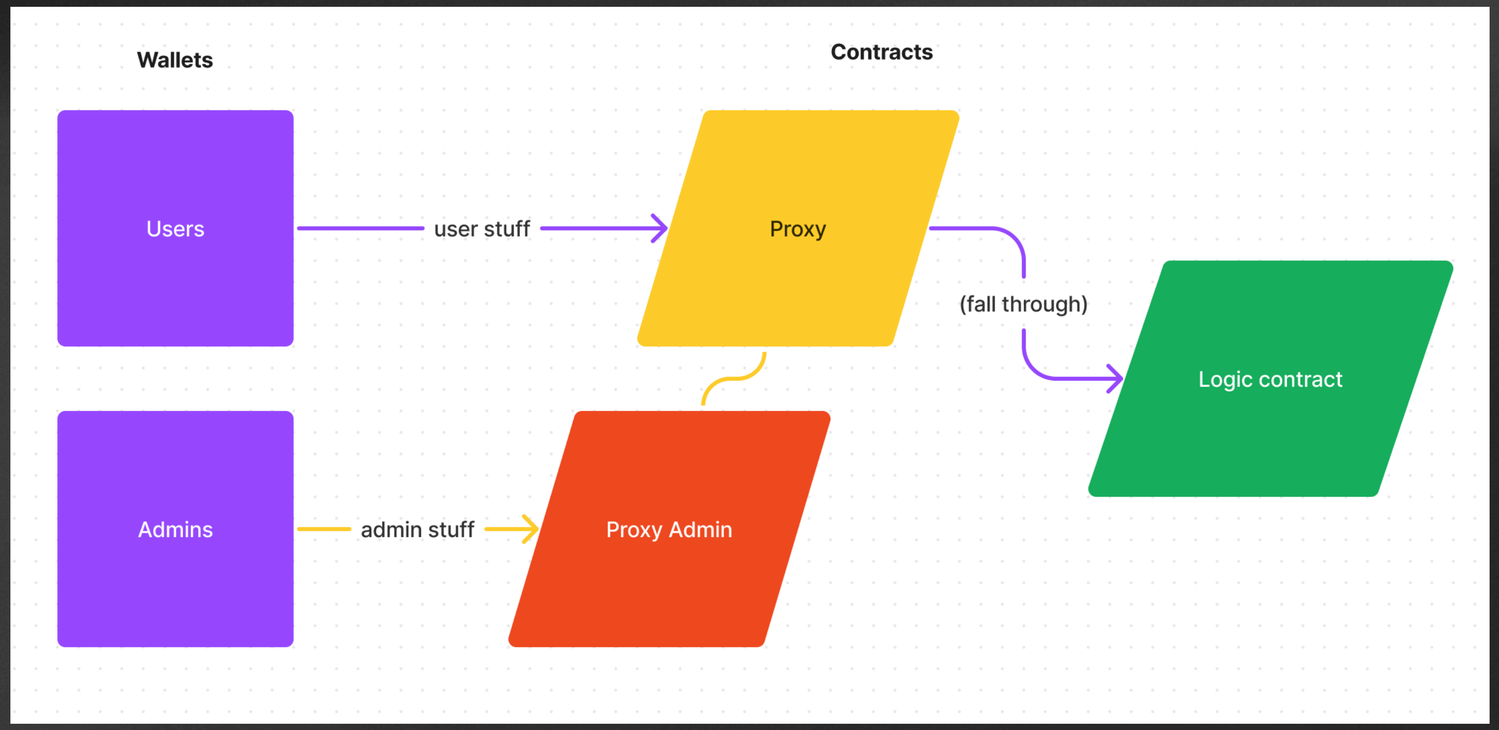
- Great for where logic is complex and needs to be reused ♻️

- Your Gnosis Safe is actually just this 👈

Deep Dive

Proxy Types #2: Transparent

- All transactions fall through to a logic contract (unless sender is admin)
- Easy to upgrade
- Must deploy an “admin” contract to manage the proxy



Proxy Types #2: Transparent

TLDR:

```
interface TransparentUpgradeableProxy {
    modifier ifAdmin() {
        if (msg.sender == admin) {
            _; // allow to execute
        } else _fallback();
    }

    function admin() external ifAdmin returns (address admin_);

    function implementation() external ifAdmin returns (address implementation_);

    function changeAdmin(address newAdmin) external virtual ifAdmin;

    function upgradeTo(address newImplementation) external ifAdmin;
}
```

Proxy Types #3: UUPS (Universal Upgradeable Proxy Standard) - ERC 1822

- A simple standard defining 1: a proxy contract 2: a base logic contract
- *All* calls are forwarded
- *No* fixed interface
- *No* admin contract / functions
- Calls are cheaper

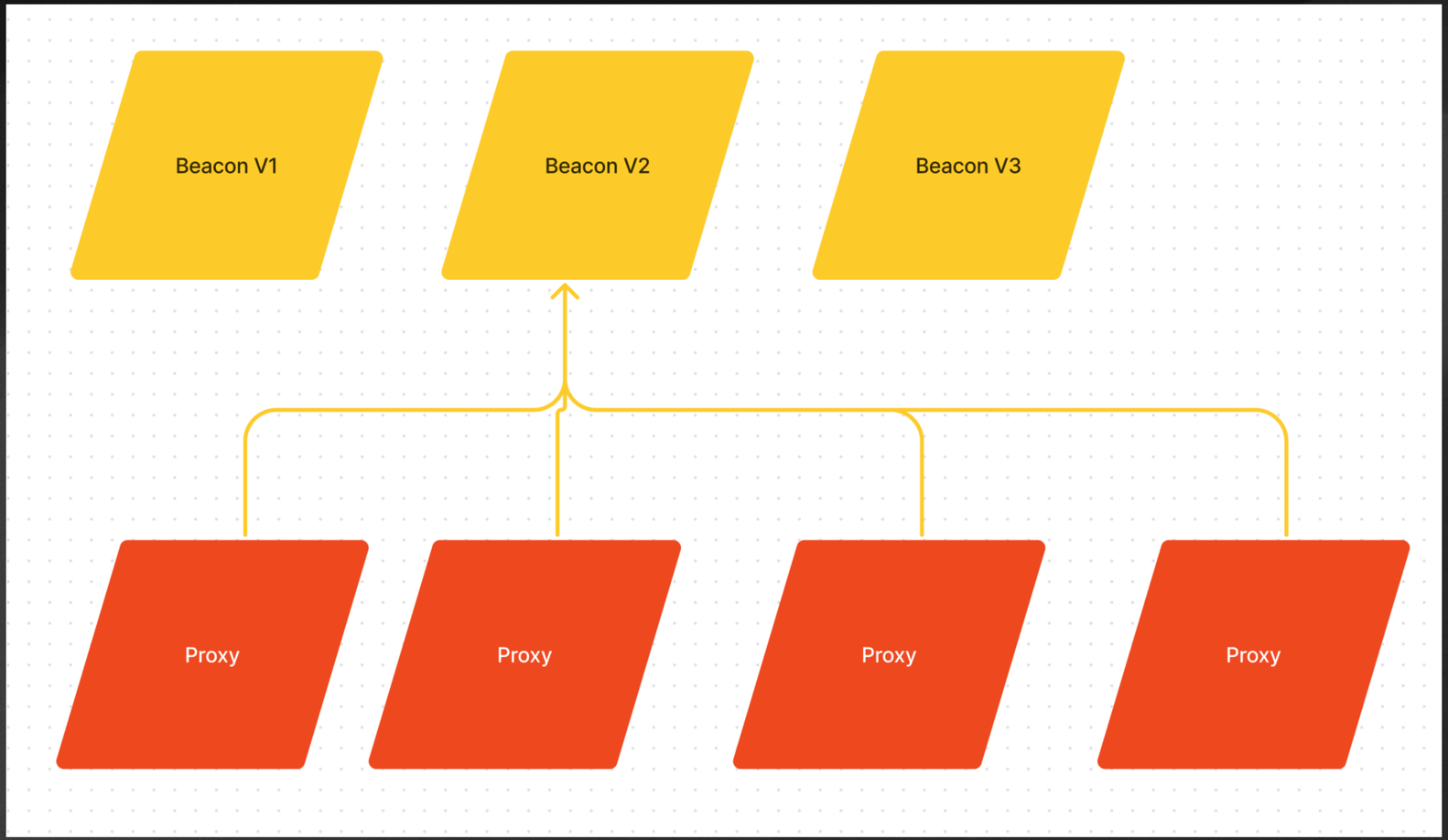
UUPS: Logic Contract TLDR

```
// 0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7
bytes32 constant IMPLEMENTATION_STORAGE_SLOT = keccak256("PROXIABLE");

abstract contract ProxiableLogicContract {
    function updateCodeAddress(address newAddress) internal {
        require(
            bytes32(IMPLEMENTATION_STORAGE_SLOT) == ProxiableLogicContract(newAddress).proxiableUUID(),
            "Not compatible"
        );
        assembly {
            sstore(IMPLEMENTATION_STORAGE_SLOT, newAddress)
        }
    }
    function proxiableUUID() public pure returns (bytes32) {
        return IMPLEMENTATION_STORAGE_SLOT;
    }
}
```

Proxy Types #4: Beacon Proxy

- Many unique contracts can be deployed but their logic can be upgraded



Proxy Survival Tips

- “master copy”, “singleton”, “implementation”, “beacon” = the contract where the logic resides

- You can’t use constructors in upgradable, proxy code 🙅🏽

- Always use the Openzeppelin hardhat plugin when developing with Proxies

- Protect your upgrade functions!

Articles

- OpenZeppelin’s Working with Upgrades

- Full Walkthrough of all Proxy Types