Alice in Proxyland

Playing with upgradeability and the Router Proxy

Alejandro Santander a.k.a. Ethernaut
Core Contributor @ Synthetix
(ex Aragon, ex OpenZeppelin)
Everyone is using proxies, but few really understand them, and their dangers.

Proper proxy usage requires tooling, and a fundamental knowledge of how they work. It’s not ok to not understand their core principles and caveats.

It’s easy tho. We’re now going to understand them by playing dumb with a silly example…

After the example, we’re going to use the core principles we learnt to talk about a new type of proxy.

Understanding how proxies work under the hood is important.
Alice and Bob deploy their first smart contract and figure out how to make it upgradeable
contract ValueHolder {
    uint value;
    address setter;

    event SetValue(uint value, address setter);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;

        emit SetValue(newValue, msg.sender);
    }
}
Alice and Bob decide to make their contract upgradeable.
contract CallProxy {
    address implementation;

    function setImplementation(address newImplementation) public {
        implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using CALL
    }
}
```
contract CallProxy {
    address implementation;

    function setImplementation(address newImplementation) public {
        implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call to the implementation using CALL
    }
}
```

```
contract ValueHolder {
    uint value;
    address setter;

    event ValueSet(uint value, address setter);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;

        emit ValueSet(newValue, msg.sender);
    }
}
```
```
contract CallProxy {
    address implementation;

    function setImplementation(address newImplementation) public {
        implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using CALL.
    }
}
```

0xb0b calls setValue(1337)

```
contract ValueHolder {
    uint value;
    address setter;

    event ValueSet(uint value, address setter);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;
        emit ValueSet(newValue, msg.sender);
    }
}
```

Event from 0x1

Storage @ 0x1
0: 1337
1: 0x2

Storage @ 0x2
0: 0x1

Execution context
Execution context?

When running EVM code, the execution context determines:

- Which contract storage to use
- Who `msg.sender` is: whoever made the call
- Which contract to emit events from
We want the execution context to be the proxy, not the implementation.

But how?
CALL vs DELEGATECALL

- **CALL** runs code in the **current** context
- **DELEGATECALL** runs code in the context of the **caller**
0xb0b calls
setImplementation(0x1)

```solidity
contract DelegateCallProxy {
    address implementation;

    function setImplementation(
        address newImplementation
    ) public {
        implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```
0xb0b calls setValue(1337)

Event from 0x3

Storage @ 0x3
0: 1337
1: 0xb0b

0x3

Execution context

DELEGATECALL

contract DelegateCallProxy {
    address implementation;
    function setImplementation( address newImplementation ) public {
        implementation = newImplementation;
    }
    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}

0x1

Storage @ 0x1
0: 1337
1: 0x2

0xb0b

contract ValueHolder {
    uint value;
    address setter;
    event SetValue(uint value, address setter);
    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;
        emit SetValue(newValue, msg.sender);
    }
}
0xb0b calls value()

Storage collision between proxy & implementation

Execution context

```solidity
contract DelegateCallProxy {
    address implementation;

    function setImplementation(
        address newImplementation
    ) public {
        implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
```
DELEGATECALL is great, but it is dangerous
How to avoid the storage collision?

Bob “destructures” the proxy’s storage.
Destructuring?

Destructuring is simply choosing where in storage to put a variable

- Solidity uses slots consecutively
- First variable is given slot 0
- Second variable slot 1, etc
- A contract has infinite potential slots
- Destructuring is just picking a custom slot
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore() private pure returns (ProxyStore storage store) {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(address newImplementation) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
0xb0b calls setValue(1337)

Execution context

contract ValueHolder {
    uint value;
    address setter;

    event ValueSet(uint value, address setter);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;

        emit ValueSet(newValue, msg.sender);
    }
}
Ok, we’ve finally found a proxy that works.

It’s time to actually use it and upgrade the implementation.
contract ValueHolderV2 {
    uint date;
    uint value;
    address setter;

    event ValueSet(uint value, address setter, uint date);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;
        date = block.timestamp;
        emit ValueSet(newValue, msg.sender, date);
    }
}
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore() private pure returns (ProxyStore storage store) {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(address newImplementation) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
0xb0b calls `value()`

Storage collision between implementations

Execution context

0xb0b is returned
Bob understands that implementation storage can only be appended to.
contract ValueHolderV3 {
    uint value;
    address setter;
    uint date;

    event SetValue(uint value, address setter, uint date);

    function setValue(uint newValue) public {
        value = newValue;
        setter = msg.sender;
        date = block.timestamp;

        emit SetValue(newValue, msg.sender, date);
    }
}
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore() private pure returns (ProxyStore storage store) {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(address newImplementation) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
Storage @ 0x4
0: 1337
1: 0xb0b
2: 0
...
1000: 0x6

1337 is returned

0xb0b calls value()
Storage collisions

It is critical to understand what they are:

- The execution context is the proxy; Everything is stored in the proxy
- Collisions can exist (1) between the proxy and the current implementation, and (2) between different versions of the implementation
- The first kind can be avoided with unstructuring the proxy’s storage layout
- The rest of the storage layout is defined by the current implementation and it can be incompatible with a previous storage layout definition
- Always append to storage
- Multiple inheritance can produce unexpected storage layouts
- It becomes hard to detect invalid mutations so tooling is needed
- Even if detected, sometimes it can be hard to avoid an invalid mutation
Why not unstructure all the storage and avoid collisions everywhere, not just in the proxy?
contract ValueHolderV4 {
    struct ValueHolderStore {
        uint value;
        address setter;
        uint date;
    }

    event ValueSet(uint value, address setter, uint date);

    function _getStore() private pure returns (ValueHolderStore storage store) {
        assembly {
            store.slot := 5080
        }
    }

    function setValue(uint newValue) public {
        ValueHolderStore storage store = _getStore();
        store.value = newValue;
        store.setter = msg.sender;
        store.date = block.timestamp;
        emit ValueSet(newValue, msg.sender, block.timestamp);
    }

    function getValue() public view returns (uint) {
        return _getStore().value;
    }
}
contract UnstructuredProxy {
    struct ProxyStore {
        address implementation;
    }

    function _getStore()
        private
        pure
        returns (ProxyStore storage store)
    {
        assembly {
            store.slot := 1000
        }
    }

    function setImplementation(
        address newImplementation
    ) public {
        _getStore().implementation = newImplementation;
    }

    fallback() external {
        // Forwards any incoming call
        // to the implementation
        // using DELEGATECALL
    }
}
Event from 0x4

```solidity
contract AccessControlProxy {
    struct AccessControl {
        address implementation;
    }

    function _authorizeCall(address sender) internal virtual returns (bool) {
        return _hasAuthority(_msgSender(), _msgData() & uint256(-1));
    }
}
```

```solidity
contract ValueHolderV4 {
    struct ValueHolderStore {
        uint value;
        address setter;
        uint date;
    }

    event ValueSet(uint value, address setter, uint date);

    function _getStore() private pure returns (ValueHolderStore storage store) {
        assembly {
            store.slot := 5000
        }
    }

    function setValue(uint newValue) public {
        ValueHolderStore storage store = _getStore();
        store.value = newValue;
        store.setter = msg.sender;
        store.date = block.timestamp;
        exit ValueSet(newValue, msg.sender, block.timestamp);
    }

    function getValue() public view returns (uint) {
        return _getStore().value;
    }
}
```

**Execution context**

- 0xb0b calls `setValue(42)`
- Storage at 0x4
  - 0: 1337
  - 1: 0xb0b
  - 2: 0
  - 1000: 0x7
  - 5000: 42
  - 5001: 0xb0b
  - 5002: 123456

**DELEGATECALL**
A proxy that works

- Execution context is kept at the proxy via DELEGATECALL
- Storage collisions are avoided by using unstructured storage everywhere
- Tooling should still be used to detect invalid storage mutations in the storage namespaces themselves
- This manual use of storage is much easier to control
Multi-contract systems and the Router Proxy
There is no ideal standard solution for multi-contract systems

Complex systems require many contracts.

These contracts need to talk to each other, and many times during these calls, they need to know if the call is coming from an official "system" contract.

People use registries, amongst other things, but in the end it all gets messy fast. And inefficient!

Let’s try a pretty crazy solution with the Router Proxy
contract AnotherContract {
  struct AnotherContractStore {
    uint coolValue;
  }

  event CoolValueSet(uint value);

  function _getStore() private pure returns (AnotherContractStore storage store) {
    assembly {
      store.slot := 9000
    }
  }

  function setCoolValue(uint newValue) public {
    AnotherContractStore storage store = _getStore();
    store.coolValue = newValue;
    emit CoolValueSet(newValue);
  }

  function getCoolValue() public view returns (uint) {
    return _getStore().coolValue;
  }
}
Bob builds and deploys a “Router”.
contract Router {
    address private constant _VALUE HOLDER = 0x8000000000000000000000000000000000000007;
    address private constant _ANOTHER CONTRACT = 0x8000000000000000000000000000000000000008;

    fallback() external payable {
        bytes sig4 = msg.sig;
        address implementation;

        assembly {
            let sig32 := shr(224, sig4)

            function findImplementation(sig) → result {
                if tt(sig, 0x0a684b) {
                    switch sig
                        case 0x20965255 { result := _VALUE HOLDER } // ValueHolder.getValue()
                        case 0x55241077 { result := _VALUE HOLDER } // ValueHolder.setValue(...)
                        leave
                    } switch sig
                        case 0x0a684b { result := _ANOTHER CONTRACT } // AnotherContract.getValuable()...
                        case 0xced15014 { result := _ANOTHER CONTRACT } // AnotherContract.setValuable()
                        leave
                } implementation := findImplementation(sig32)
            }

            assembly {
                // Forwards any incoming call
                // to the implementation
                // using DELEGATECALL
            }
        }
    }
}
Bob sets the router as the proxy’s implementation.
contract UnstructuredProxy {
  struct ProxyStore {
    address implementation;
  }

  function _getStore() public pure returns (ProxyStore storage store) {
    assembly {
      store.slot := 1000
    }
  }

  function setImplementation( address newImplementation ) public {
    _getStore().implementation = newImplementation;
  }

  fallback() external {
    // Forwards any incoming call to the implementation using DELEGATECALL
  }
}
Event from 0x4

Storage @ 0x4
0: 1337
1: 0xb0b
2: 0
...
5000: 42
5001: 0xb0b
5002: 123456
...
9000: 7

0xb0b calls
setCoolValue(7)
Bob upgrades one of the system’s modules.
contract AnotherContractV2 {
  struct AnotherContractStore {
    uint coolValue;
  }
  event CoolValueSet(uint value);
  function _getStore() private pure returns (AnotherContractStore storage store) {
    assembly {
      store.slot := 9800
    }
  }
  function setCoolValue(uint newValue) public {
    AnotherContractStore storage store = _getStore();
    store.coolValue = newValue * 7;
    emit CoolValueSet(newValue);
  }
  function getCoolValue() public view returns (uint) {
    return _getStore().coolValue;
  }
}
contract RouterV2 {
    address private constant _VALUE_HOLDER = 0x8965255;
    address private constant _ANOTHER_CONTRACT = 0x55241877;

    fallback() external payable {
        bytes4 sig4 = msg.sig;
        address implementation;

        assembly {
            let sig32 := shr(224, sig4)

            function findImplementation(sig) -> result {
                if lt(sig, 0x8965255) {
                    switch sig
                        case 0x8965255 { result := _VALUE_HOLDER } // Holder.getValue()
                        case 0x55241877 { result := _VALUE_HOLDER } // Holder.setValue(...)
                        leave
                    }
                    switch sig
                        case 0x8965255 { result := _ANOTHER_CONTRACT } // AnotherContract.getCoolValue()
                        case 0x55241877 { result := _ANOTHER_CONTRACT } // AnotherContract.setCoolValue()
                        leave
                }
                implementation := findImplementation(sig32)
            }

            assembly {
                // Forwards any incoming call
                // to the implementation
                // using DELEGATECALL
            }
        }
    }
}
contract UnstructuredProxy {
  struct ProxyStore {
    address implementation;
  }

  function _getStore() private pure
    returns (ProxyStore storage store)
  {
    assembly {
      store.slot := 1000
    }
  }

  function setImplementation( 
    address newImplementation
  ) public {
    _getStore().implementation = newImplementation;
  }

  fallback() external {
    // Forwards any incoming call
    // to the implementation
    // using DELEGATECALL
  }
}
What would a more complex system look like?
Unused storage

Each system module is a contract, and thus has its own storage. However, such storage is not used and completely ignored in this architecture.
Gas efficiency

- A transparent proxy uses ~3000 gas
- A UUPS proxy uses ~1600 gas
- This proxy and router uses ~2600 gas
Inter-modular communications

- All modules are the system
- Cast and call self?
  - from ModuleB: ModuleA(this).doSomething()
  - but msg.sender is lost
- Cast and delegatecall self?
  - works
- Mixins
  - Pieces of code that know how to interact with storage
  - Not deployed, inherited
  - Act as delegates of another module
  - Avoids calls altogether!
  - Communication becomes super cheap
Mixin example
contract OwnerStorage {
    struct OwnerStore {
        address owner;
    }

    function _getOwnerStore() internal pure returns (OwnerStore storage store) {
        assembly {
            store.slot := 231234
        }
    }
}
contract OwnerMixIn is OwnerStorage {
    error Unauthorized(address who);

    modifier onlyOwner() {
        OwnerStore storage store = _getOwnerStore();

        if (msg.sender != store.owner) {
            revert Unauthorized(msg.sender);
        }
    }
}
contract OwnerModule is OwnerMixin {
    function owner(address newOwner) public onlyOwner {
        OwnerStore storage store = _getOwnerStore();
        store.owner = newOwner;
    }

    function getOwner() internal view returns (address) {
        return _getOwnerStore().owner;
    }
}
contract ValueHolderVS is OwnerMixin {
  struct ValueHolderStore {
    uint value;
    address setter;
    uint date;
  }
  
  event ValueSet(uint value, address setter, uint date);
  
  function _getStore() private pure returns (ValueHolderStore storage store) {
    assembly {
      store.slot := 5000
    }
  }

  function setValue(uint newValue) public onlyOwner {
    ValueHolderStore storage store = _getStore();
    store.value = newValue;
    store.setter = msg.sender;
    store.date = block.timestamp;
    emit ValueSet(newValue, msg.sender, block.timestamp);
  }

  function getValue() public view returns (uint) {
    return _getStore().value;
  }
}
Slight code style change

- Can’t use “regular” Solidity variables
- Must use storage namespaces always
- It’s not as weird as it looks, and you get used to it fast
- Should Solidity do this under the hood?
  - Proposed by maxsam4:
    https://github.com/ethereum/solidity/issues/11102
Why use the router?

- No more contract size limits!
  - All contracts are “merged” into one
- Simplified and efficient inter-modular communications
  - Ideal for complex systems
- System composition is explicitly specified in a single contract and upgraded with a single tx
  - Ideal for governance
- Core component of Synthetix v3
How to use the router?

- Hardhat plugin @
- Generates the router
- Manages storage namespaces
- Performs validations to ensure that there are no storage collisions
Thank you!

Alejandro Santander a.k.a. Ethernaut
Core Contributor @ Synthetix

@the_ethernaut