Contract upgrade risks and remediations

Who Am I

- Josselin Feist, josselin@trailofbits.com

- Trail of Bits: trailofbits.com
  - We help organizations build safer software
  - R&D focused: we use the latest program analysis techniques

Goals

- What is contract upgradability?
- Existing solutions
- Existing alternative
- Conclusion

Upgradability

Smart contracts

- ‘Code is law’
- Ethereum smart contracts are immutables by design

Immutability

- **Great for users**
  - Minimize trust require by contract's users
  - Users can check the set of rules

- **Require caution for developers**
  - New features?
  - Bug fixes?

Contract Upgradability

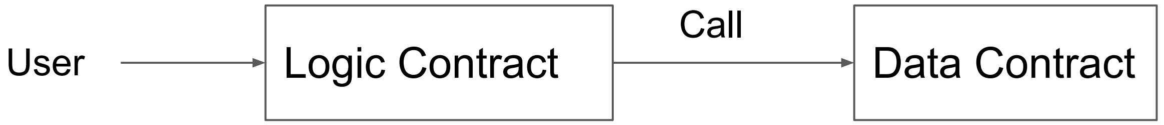
- **Solution: Add upgradability capacity**
- **Two main strategies:**
    - Data Separation
    - Delegatecall Proxy

https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/

Data Separation

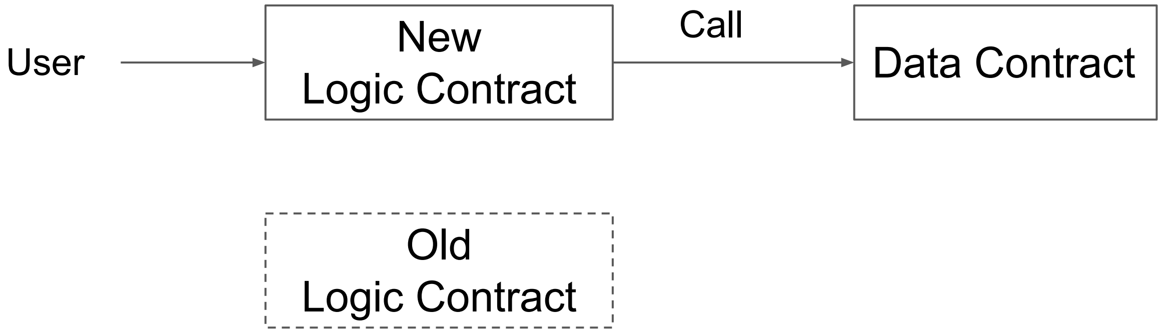
Data Separation

- **Two contracts:**
  - Logic: holds the logic (mutable)
  - Data: holds the data (immutable)

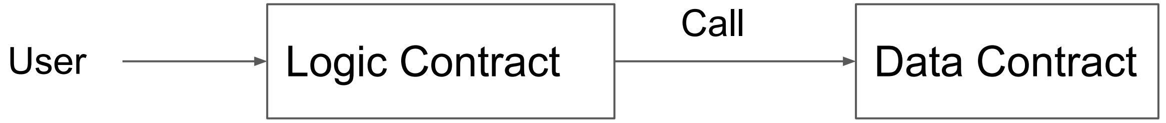


How to Upgrade

- **Upgrade: deploy new logic contract**



Data Separation

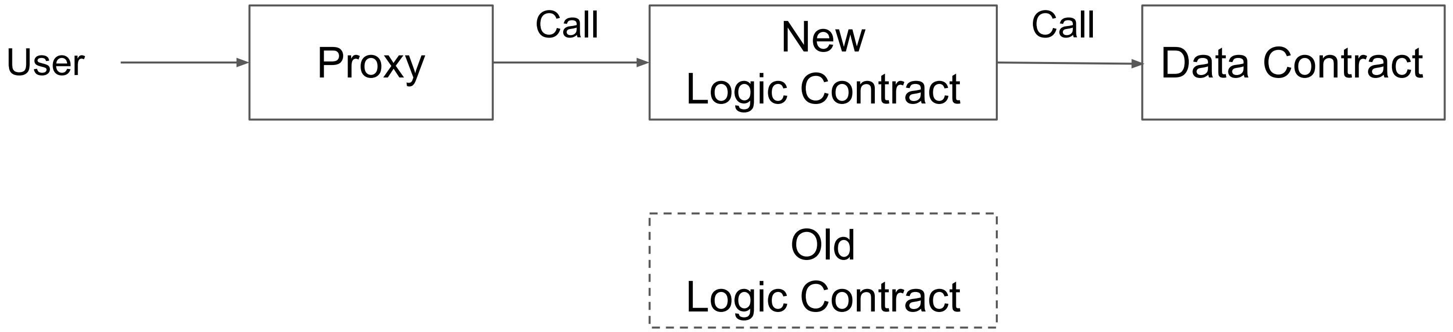


```
contract Logic{
  Data data;

  function inc() public{
    data.setV(data.v() + 1);
  }

  function v() public returns(uint){
    return data.v();
  }
```

```
contract Data is Owner {
  uint public v;
  function setV(uint new_v) onlyOwner
public {
    v = new_v;
 }
```

Data Separation: logic alternative

- **Use of a third proxy contract**
  - Provide constant entry point for users



Data Separation: Recommendations

- **Define clear separation between data and logic**
- **Keep simple implementation**
  - Avoid complex data storage (ex: key-value pair)
- **Define the upgrade procedure**
  - How to upgrade the contracts? Pause contracts?
  - How to store the keys?
  - How to communicate with the users?

Delegatecall Proxy

EVM Internals

- **EVM has a harvard architecture**
  - Section code != Section data
- **A contract can**
  - Call another contract
- **A contract cannot**
  - Write directly to another contract’s data

delegatecall instruction

- **Delegatecall instruction:**
  - Execute code from external contracts from the caller’s data context
- **Example:**
  - contract A *delegetecalls* to contract B
  - The code of contract B will be executed using the data of contract A
- **Designed for libraries**

Library

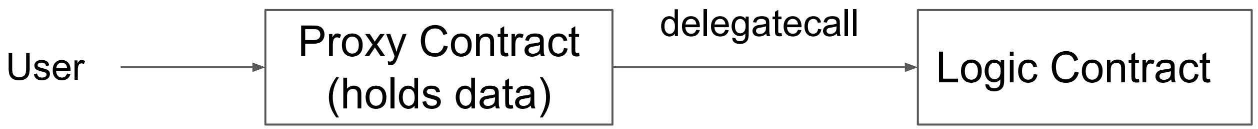
```
library Lib {
  struct Data { uint val; }
  function set(Data storage self, uint new_val) public {
    self.val = new_val;
  }
}

contract C {
  Lib.Data public myVal;
  function set(uint new_val) public {
    Lib.set(myVal, new_val);
  }
}
```

delegatecall

- **Callee needs to know the exact memory layout of the caller**
  - For library: handled by the compiler
  - For user-level call: needs to be **really** careful

Upgradability through delegatecall

- **Two contracts:**
  - Proxy contract: holds data: (immutable)
  - Logic: holds the logic (mutable)
  - Fallback function of Proxy delegatecalls to Logic

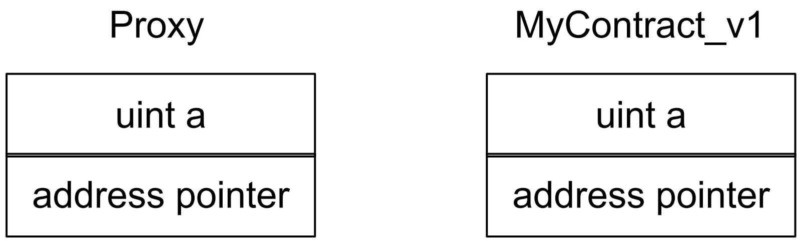


Upgradability through delegatecall

- **Upgrade: change the logic contract**
- **Each version of the logic contract must follow the same memory layout**
  - Do you know precisely how Solidity store variables in memory?

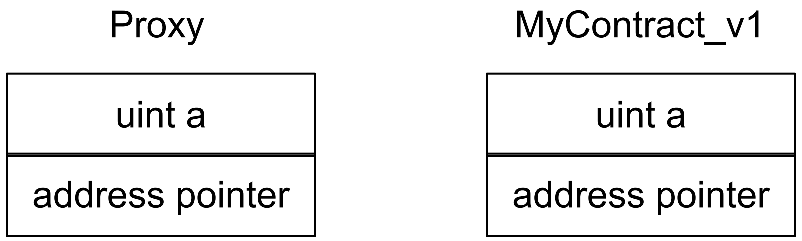
Delegatecall Example

```
contract Proxy {
 uint public a;
 address public pointer;

 ...

 function () public {
   pointer.delegatecall(..)
 }
}

contract MyContract_v1 {
 uint public a;
 address public pointer;

 function set(uint val) public {
    a = val;
 }
}
```



Delegatecall Example

```
contract Proxy {
 uint public a;
 address public pointer;

 ...

 function () public {
   pointer.delegatecall(..)
 }
}
```

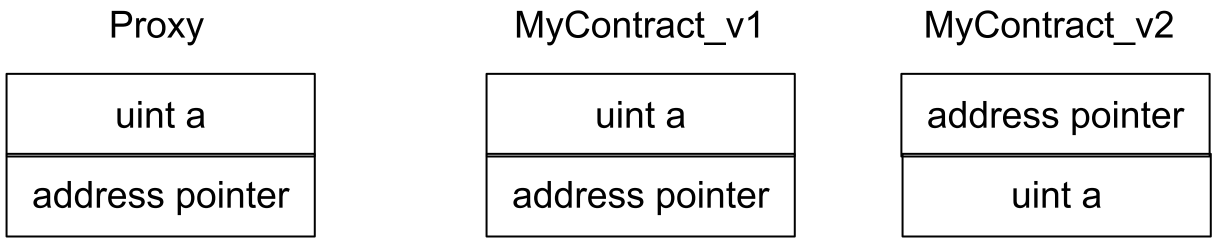


```
contract MyContract_v1 {
 uint public a;
 address public pointer;

 function set(uint val) public {
    a = val;
 }
}
```

```
contract MyContract_v2 {
 address public pointer;
 uint public a;

 function set(uint val) public {
    a = val;
 }
}
```

Delegatecall Example

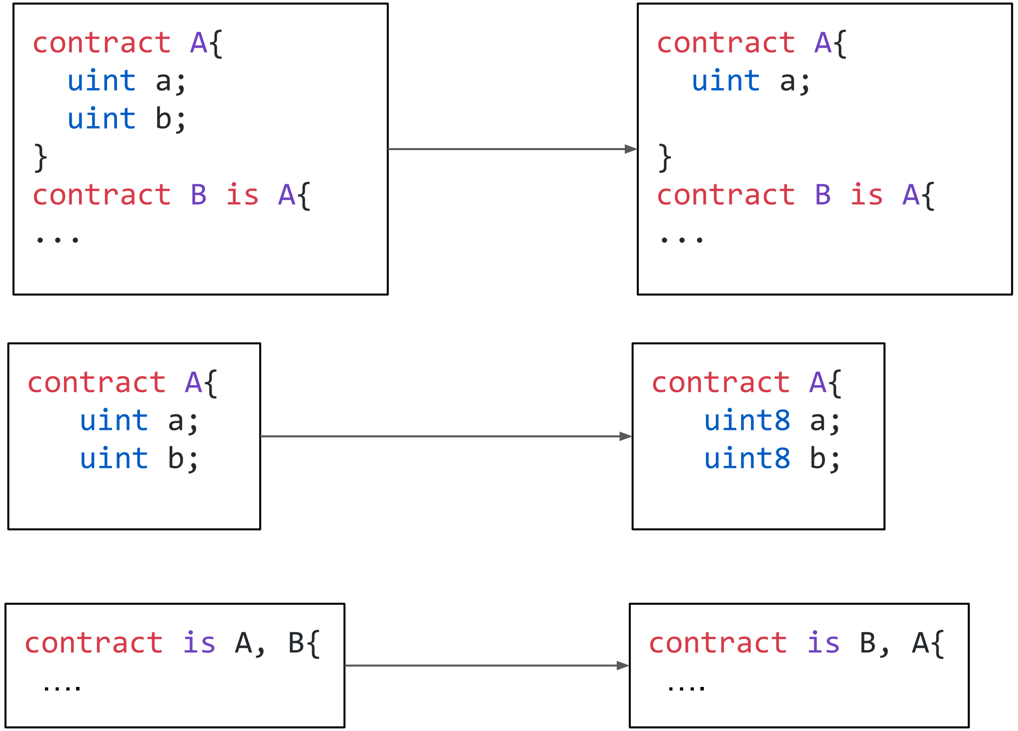
```
contract Proxy {
 uint public a;
 address public pointer;

 ...

 function () public {
   pointer.delegatecall(..)
 }
}
```



```
contract MyContract_v1 {
 uint public a;
 address public pointer;

 function set(uint val) public {
    a = val;
 }
}
```

```
contract MyContract_v2 {
 address public pointer;
 uint public a;

 function set(uint val) public {
    a = val;
 }
}
```

Examples of incorrect upgrades



```
contract A{
  uint a;
  uint b;
}
contract B is A{
...
```

```
contract A{
  uint a;

}
contract B is A{
...
```

```
contract A{
   uint a;
   uint b;
```

```
contract A{
   uint8 a;
   uint8 b;
```

```
contract is A, B{
 ....
```

```
contract is B, A{
 ....
```

Delegatecall Proxy: Recommendations

- **Never remove a variable**
- **Never change a variable type**
  - Packing issue
- **Be careful with inheritance**
  - Inheritance order impacts the memory layout

Delegatecall Proxy: Recommendations

- **Use same compiler version**
  - Solidity could have better optimizations
- **Be careful with correct contract initialization**
  - Constructors cannot be used
- **Inspect the generated EVM code manually**
  - No mature tool can validate the memory layout

Delegatecall Proxy: Recommendations

- **Define the upgrade procedure**
  - How to upgrade the contracts?
  - How to store the keys?
  - How to communicate with the users?

Upgradability: Summary

Upgradability: Summary

- **Allow to patch contracts**

- **Drawbacks for developers:**
  - Increase code size and complexity
  - Require extra knowledge
  - Increase the number of keys
  - Encourage solving problems after deployments
- **Drawbacks for users:**
  - Increase gas cost
  - Prevent trustless system

Upgradability: recommendations

- **Be careful when choosing features of your contract**
- **Strive for the simplest solution**
    - Use data separation over deletegacalll
- **Don’t add upgradability at the end of our development process**

Alternative?
Contract Migration

Contract Migration

- Copy variables from the contract to a new version

Why do you need a Migration?

- **To upgrade non-upgradable contracts**
- **To recover from a compromise**
    - Contract compromise
    - Key(s) compromise
- **To recover from incorrect setup**
    - owner = 0

How to perform a migration?

1. **Data recovery: Collect the values of the variables**
   - Use Events
2. **Data writing: Deploy and initiate the new contract**
   - Use an initialization state
   - Migration 300.000 balances = $7,500 in October

See recommendations: How Contract Migration Works

https://blog.trailofbits.com/2018/10/29/how-contract-migration-works/

Migration versus Upgradability

- **Migration covers most of the benefits of upgradability**
- **Arguments for migration:**
  - Simple than upgradability
  - No additional code
  - No additional key
  - No additional trust from the user
  - No additional cost for users

Upgradable contracts also need a migration procedure

Migration versus Upgradability

- **Arguments for upgradability:**
  - Frequent update (cost of migration)
    - One migration is cheap (see blogpost)
  - Fixed address required

Takeaway

- **Be prepared to migrate your contract**
  - Reacting quickly after a compromise is error prone
- **Evaluate if you need upgradability in addition to migration**
- **Hire security experts**

More information:

- https://blog.trailofbits.com/2018/09/05/contract-upgrade-anti-patterns/
- https://blog.trailofbits.com/2018/10/29/how-contract-migration-works/

TRAIL OF BITS