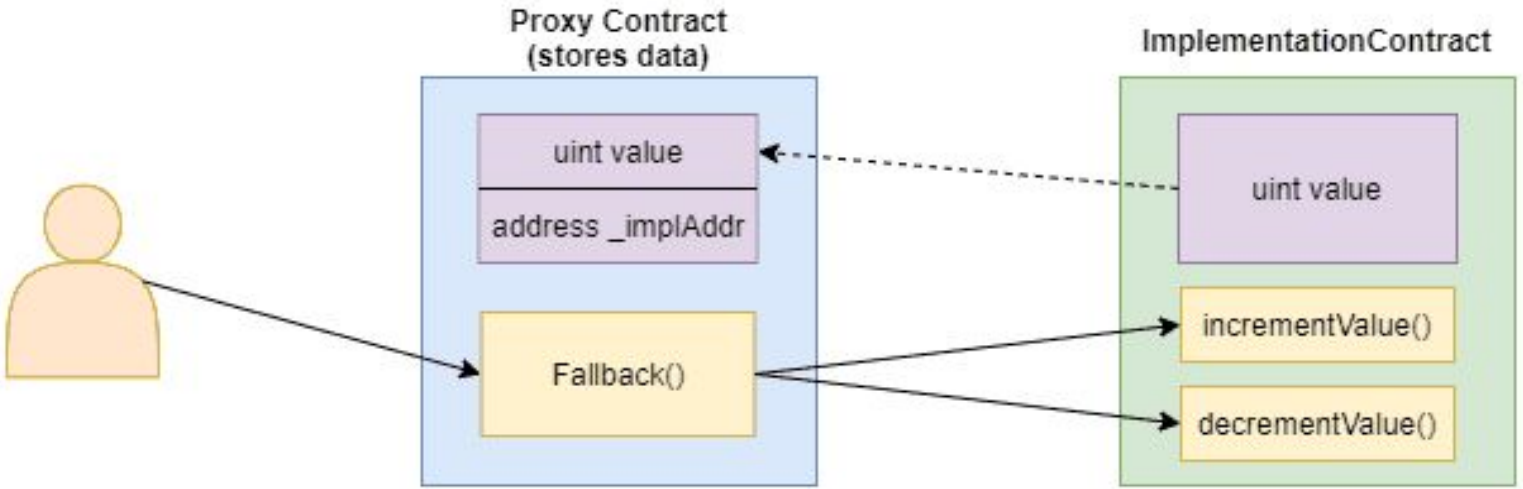
Better Detection of Upgradable Proxy Contracts in Ethereum with Slither

Bill Bodell, Sajad Meisami
September 22, 2021

Overview: Upgradable Smart Contracts

- Ethereum is a blockchain platform that features a Turing-complete virtual machine (EVM) on which programs called smart contracts are stored and executed
- As with most blockchains, smart contract code stored on Ethereum is immutable
  - This is a feature, not a bug
  - However, this means that bugs discovered in deployed contracts cannot be patched
- Therefore, the Ethereum community has developed a number of design patterns for implementing upgradable smart contracts, most commonly involving a proxy contract
  - The proxy contract is immutable, and stores the address of a separate logic contract which can be updated, along with its balance and the values of all other state variables

Overview: Upgradable Smart Contracts



```
// This code is a simplified example of a proxy contract. DO NOT USE IN THE REAL WORLD.
contract ExampleProxy {
   address delegate;
   address owner = msg.sender;

   function upgradeDelegate(address newDelegateAddress) public {
       require(msg.sender == owner);
       delegate = newDelegateAddress;
   }

   fallback() external payable {
       assembly {
           let _target := sload(0)
           calldatacopy(0x0, 0x0, calldatasize())
           let result := delegatecall(gas(), _target, 0x0, calldatasize(), 0x0, 0)
           returndatacopy(0x0, 0x0, returndatasize())
           switch result case 0 {revert(0, 0)} default {return (0, returndatasize())}
       }
   }
}
```

Overview: Slither

- Slither is a static analysis framework, written in Python, for evaluating and discovering vulnerabilities in Ethereum smart contracts written in Solidity
- It runs a suite of vulnerability detectors, prints visual information about contract details, and provides an API to easily write custom analyses
- Slither translates Solidity to an intermediate representation, SlithIR, to enable high-precision analysis via a simple API. It supports taint and value tracking to enable detection of complex patterns.

https://github.com/crytic/slither

Overview: Slither

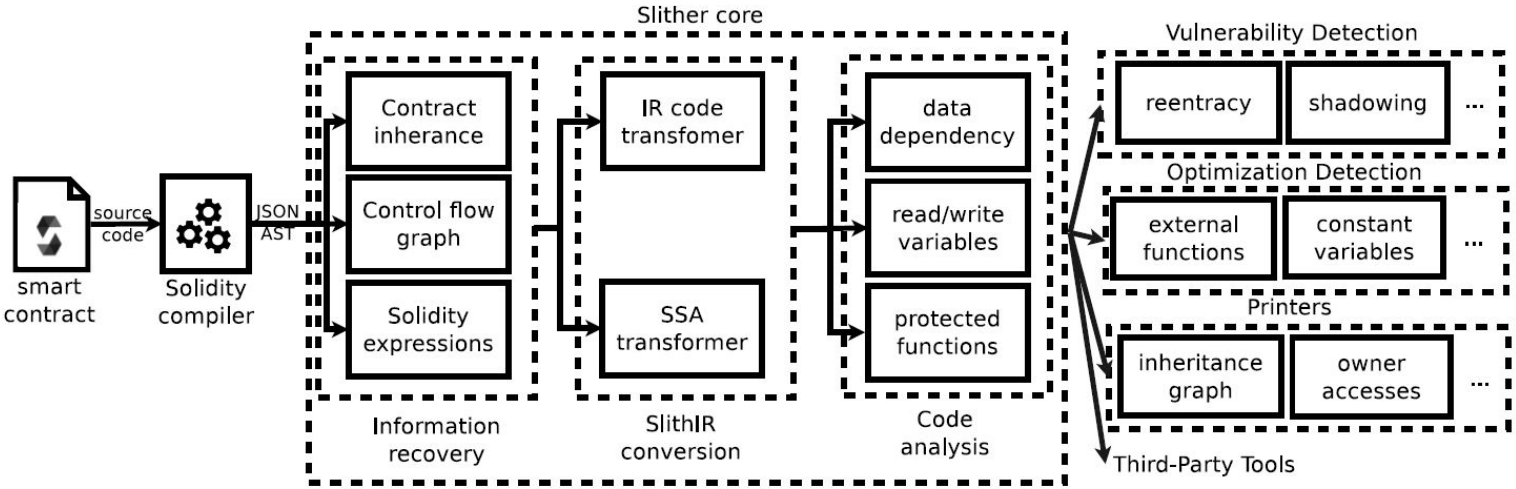


Fig. 1: Slither overview

Slither: A Static Analysis Framework For Smart Contracts, Josselin Feist, Gustavo Grieco, Alex Groce - WETSEB '19

```
webthe3rd:~/../VNFTx$ slither VNFTx.sol
INFO:Detectors:
VNFTx.buyAddon(uint256,uint256) (VNFTx.sol#280-308) ignores return value by
muse.transferFrom(msg.sender,_addon.artistAddr,artistCut) (VNFTx.sol#305)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#unchecked-transfer
INFO:Detectors:
VNFTx.challengesUsed (VNFTx.sol#138) is never initialized. It is used in:
        - VNFTx.getChallenges(uint256) (VNFTx.sol#272-278)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#uninitialized-state-variables
INFO:Detectors:
Reentrancy in VNFTx.editAddon(uint256,string,uint256,uint256,uint256,string,address,bool) (VNFTx.sol#440-472):
        External calls:
        - addons.mint(address(this),_id,_quantity.sub(_addon.quantity),) (VNFTx.sol#460)
        - addons.burn(address(this),_id,_addon.quantity - _quantity) (VNFTx.sol#462)
        State variables written after the call(s):
        - _addon.quantity = _quantity (VNFTx.sol#464)
        - _addon.used = _used (VNFTx.sol#465)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#reentrancy-vulnerabilities-1
INFO:Detectors:
VNFTx.setHealthStrat(uint256,uint256,uint256,uint256,uint256,uint256) (VNFTx.sol#486-508) should emit an event for:
        - hpMultiplier = _hpMultiplier (VNFTx.sol#502)
        - rarityMultiplier = _rarityMultiplier (VNFTx.sol#503)
        - addonsMultiplier = _addonsMultiplier (VNFTx.sol#505)
        - expectedRarity = _expectedRarity (VNFTx.sol#506)
Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-events-arithmetic
```

Overview: Slither

What is an intermediate representation (IR)?

- In language design, compilers often operate on an “intermediate representation” (IR) of a language that carries extra details about the program as it is parsed (i.e. a parse tree)
- The compiler can continue to enrich this tree with information, such as taint information, source location, and other items that could have impacted an item from control flow
- Languages such as Solidity have inheritance, meaning that functions and methods may be defined outside the scope of a given contract
    - An IR could linearize these methods, allowing additional transformations and processing of the contract’s source code

https://github.com/crytic/slither/wiki/SlithIR

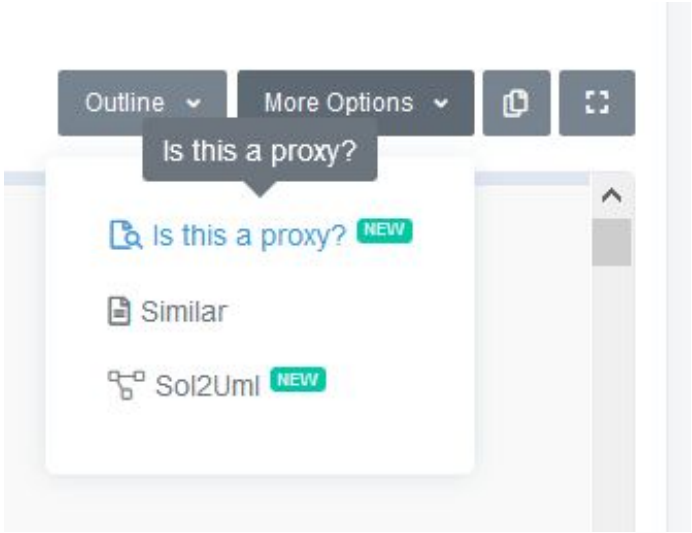
Overview: Slither

- Slither is a very well-written and well-maintained open-source project, which is popular in the industry due in part to its easy of use and wide coverage of known vulnerabilities
- Along with the set of detectors that come with the Slither tool, the repo also contains additional tools which build upon Slither's core functionality, including:
  - `slither-check-upgradeability`: Review `delegatecall`-based upgradeability
  - `slither-prop`: Automatic unit test and property generation
  - `slither-flat`: Flatten a codebase
  - `slither-check-erc`: Check the ERC's conformance
  - `slither-format`: Automatic patch generation

Overview: Etherscan

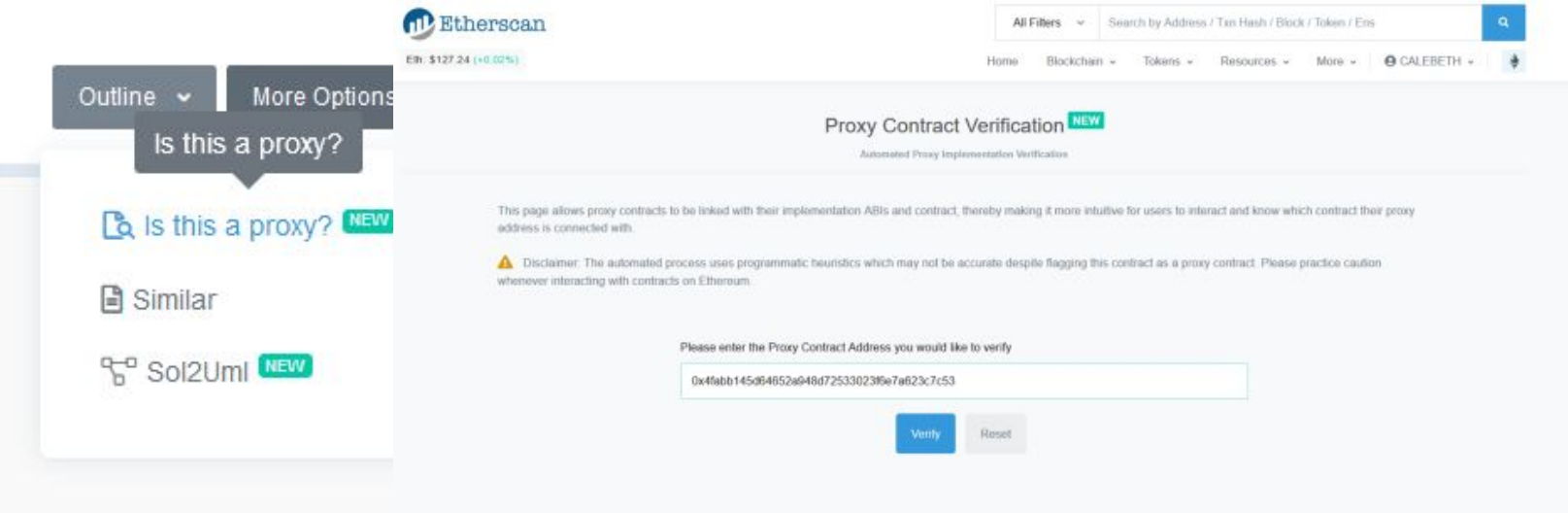
- Etherscan is the most popular block explorer used by the Ethereum community to browse the current state and history of the blockchain, accounts, transactions, contracts, etc.
- Owing to its popularity, Etherscan has also become a repository for verified smart contract source code, equipped with features for both analyzing and interacting with live contracts
- Toward the end of 2019, Etherscan released an experimental tool meant to make it easier to interact with upgradable contracts using a proxy standard
    - The usual Read/Write Contract feature doesn't support cross-contract interaction
    - The new tool tries to identify proxies by looking for the `delegatecall` opcode, then runs a number of checks to try to obtain the implementation contract

Overview: Etherscan



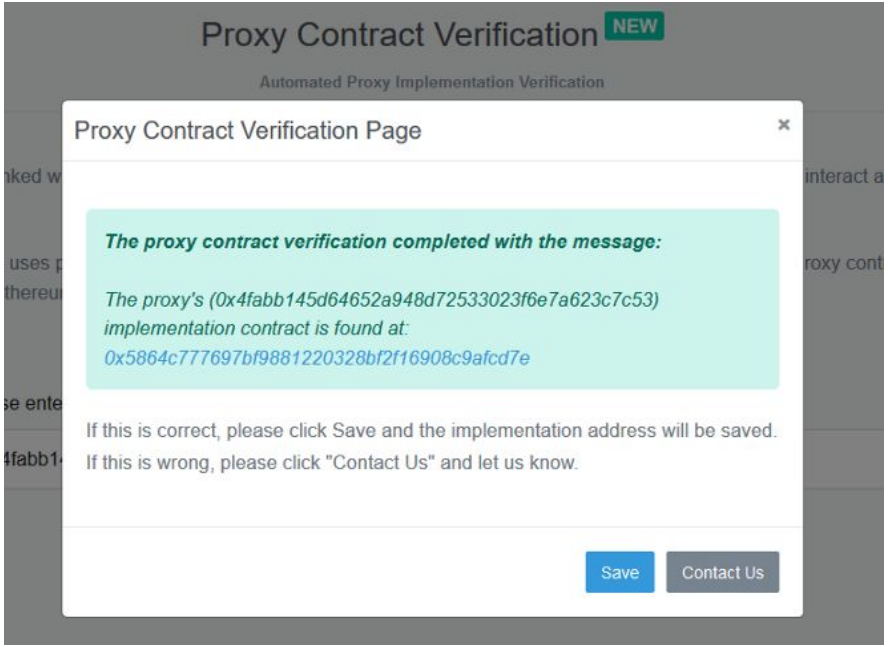
https://medium.com/etherscan-blog/and-finally-proxy-contract-support-on-etherscan-693e3da0714b

Overview: Etherscan



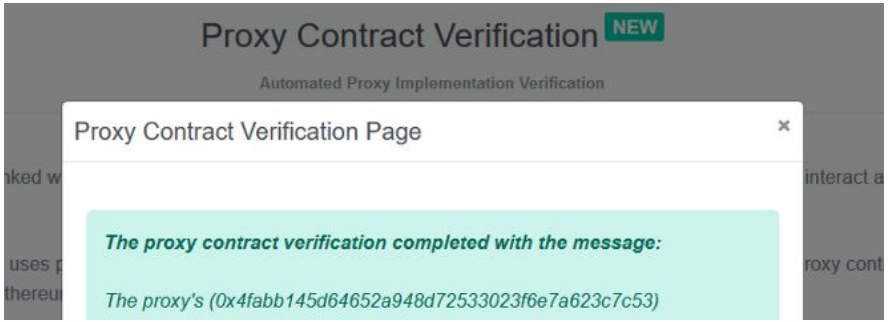
https://medium.com/etherscan-blog/and-finally-proxy-contract-support-on-etherscan-693e3da0714b

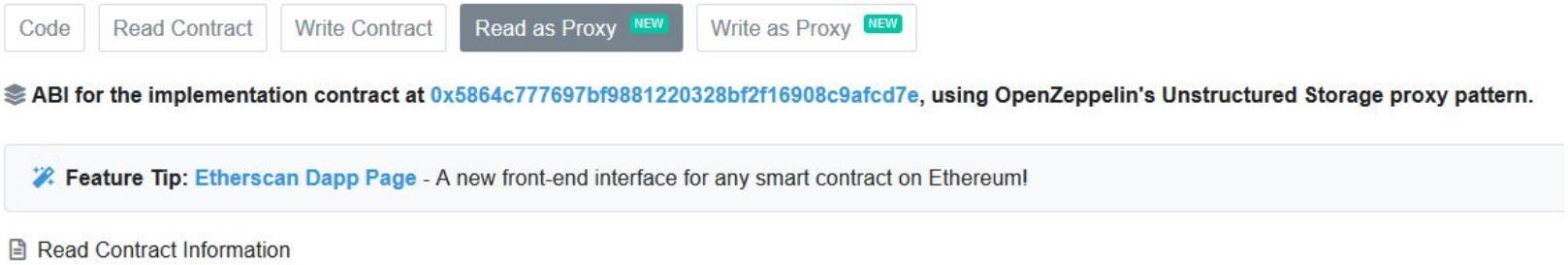
Overview: Etherscan



https://medium.com/etherscan-blog/and-finally-proxy-contract-support-on-etherscan-693e3da0714b

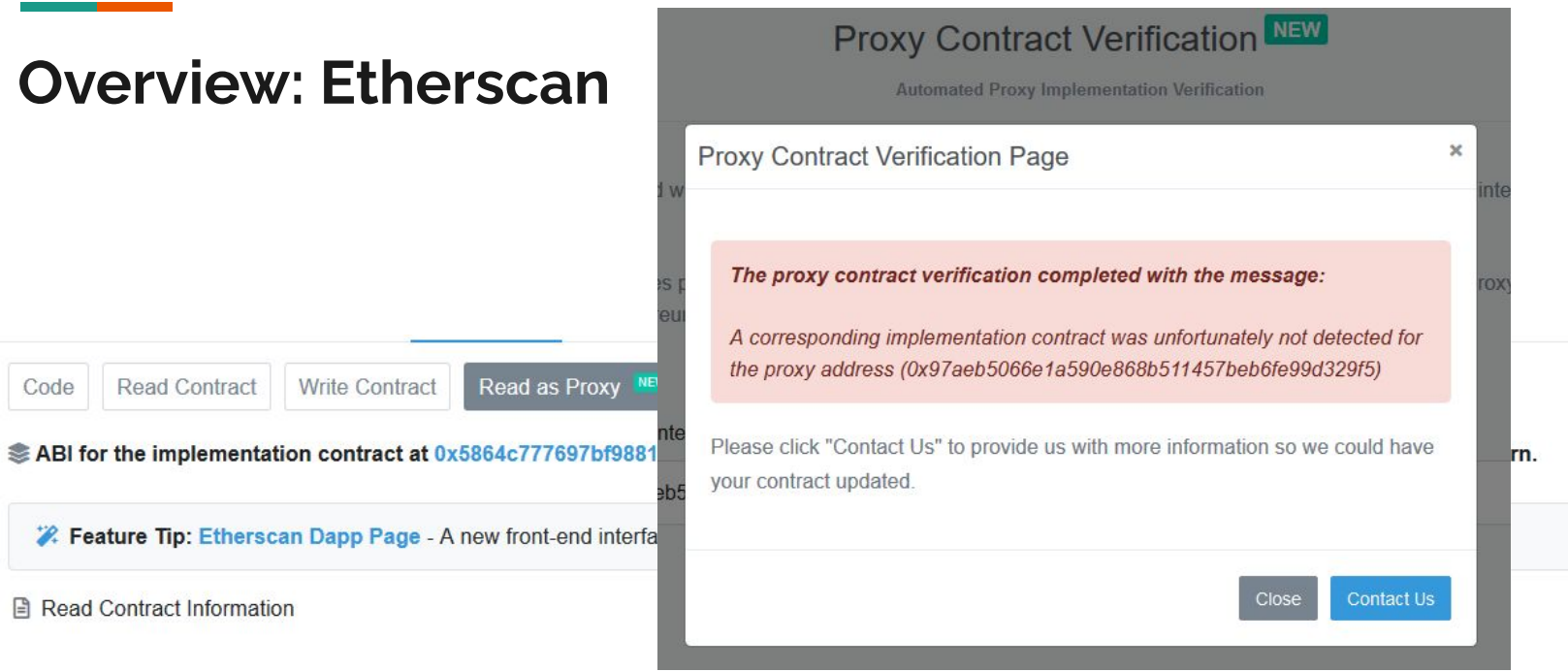
Overview: Etherscan





https://medium.com/etherscan-blog/and-finally-proxy-contract-support-on-etherscan-693e3da0714b

Overview: Etherscan



https://medium.com/etherscan-blog/and-finally-proxy-contract-support-on-etherscan-693e3da0714b

The Problem

- The `is_upgradeable_proxy()` property of the Slither class `Contract` is broken, causing both **False Positives** and **False Negatives**
- Naively checking for the string 'Proxy' in the contract's name causes **False Positives**, i.e. when analysing a contract called `ProxyFactory`, which creates Proxies but is not one itself
- Removing that check, the remainder of the method causes **False Negatives** due to inconsistencies in how functions are parsed during solc parsing
- This `Contract.is_upgradeable_proxy()` property is used by `Contract.is_upgradeable`, i.e. if a contract is an upgradable proxy contract, then it can't be an upgradable *logic* contract

The Problem

- Etherscan's proxy tool is not perfect either, but it seems to work more consistently than the pre-existing Slither checks, at least with proxy contracts that conform to a standard pattern
- However, using verified smart contract source code scraped from Etherscan, we discovered a number of non-standard proxies which yielded **False Negatives** using their tool
- Furthermore, many of the upgradable proxy contracts we tested on Etherscan had been updated to a logic contract which has not been uploaded for verification, meaning the tool cannot interact with it via the proxy

Our Goals

- Through developing a deep understanding of the upgradable proxy pattern's many variants, we intend to drastically improve detection of proxy contracts and their implementations in Slither, systematically tracking down and eliminating false positives and false negatives
  - This will need to include cross-contract analysis, i.e. to catch cases where the proxy is managed by another contract which contains the code for updating the proxy's logic
- Once we have completed our fixes in Slither, we will assess the effectiveness of our new upgradability checks by comparing our results to those of the original algorithm, as well as with Etherscan's proxy detection tool
  - With this done, perhaps we can collaborate with Etherscan to improve their tool

Questions?

Edge Cases in Detecting Proxy Contracts with Slither: Part 1

Bill Bodell
July 20, 2021

```python
@property
def is_upgradeable(self) -> bool:
    if self._is_upgradeable is None:
        self._is_upgradeable = False
        if self.is_upgradeable_proxy:
            return False
        initializable = self.compilation_unit.get_contract_from_name("Initializable")
        if initializable:
            if initializable in self.inheritance:
                self._is_upgradeable = True
        else:
            for c in self.inheritance + [self]:
                # This might lead to false positive ← original author's comment
                lower_name = c.name.lower()
                if "upgradeable" in lower_name or "upgradable" in lower_name:
                    self._is_upgradeable = True
                    break
                if "initializable" in lower_name:
                    self._is_upgradeable = True
                    break
    return self._is_upgradeable
```

The `Contract.is_upgradeable()` method (also needs a lot of work, but we'll get there when we get there)

```python
@property
def is_upgradeable_proxy(self) -> bool:
    from slither.core.cfg.node import NodeType
    from slither.slithir.operations import LowLevelCall

    if self._is_upgradeable_proxy is None:
        self._is_upgradeable_proxy = False
        if "Proxy" in self.name:
            self._is_upgradeable_proxy = True
            return True
        for f in self.functions:
            if f.is_fallback:
                for node in f.all_nodes():
                    for ir in node.irs:
                        if isinstance(ir, LowLevelCall) and ir.function_name == "delegatecall":
                            self._is_upgradeable_proxy = True
                            return self._is_upgradeable_proxy
                    if node.type == NodeType.ASSEMBLY:
                        inline_asm = node.inline_asm
                        if inline_asm:
                            if "delegatecall" in inline_asm:
                                self._is_upgradeable_proxy = True
                                return self._is_upgradeable_proxy
    return self._is_upgradeable_proxy
```

The original, broken `Contract.is_upgradeable_proxy` method

```python
@property
def is_upgradeable_proxy(self) -> bool:
    from slither.core.cfg.node import NodeType
    from slither.slithir.operations import LowLevelCall

    if self._is_upgradeable_proxy is None:
        self._is_upgradeable_proxy = False



        for f in self.functions:
            if f.is_fallback:
                for node in f.all_nodes():
                    for ir in node.irs:
                        if isinstance(ir, LowLevelCall) and ir.function_name == "delegatecall":
                            self._is_upgradeable_proxy = True
                            return self._is_upgradeable_proxy
                    if node.type == NodeType.ASSEMBLY:
                        inline_asm = node.inline_asm
                        if inline_asm:
                            if "delegatecall" in inline_asm:
                                self._is_upgradeable_proxy = True
                                return self._is_upgradeable_proxy
    return self._is_upgradeable_proxy
```

Without the check for “Proxy” in the name, we are left with two faulty checks on the fallback function

```python
@property
def is_upgradeable_proxy(self) -> bool:
    from slither.core.cfg.node import NodeType
    from slither.slithir.operations import LowLevelCall

    if self._is_upgradeable_proxy is None:
        self._is_upgradeable_proxy = False



        for f in self.functions:
            if f.is_fallback:
                for node in f.all_nodes():
                    for ir in node.irs:
                        if isinstance(ir, LowLevelCall) and ir.function_name == "delegatecall":
                            self._is_upgradeable_proxy = True
                            return self._is_upgradeable_proxy







    return self._is_upgradeable_proxy
```

This first check does not trigger on the most basic proxy if the `delegatecall` is within `assembly{...}`

```python
@property
def is_upgradeable_proxy(self) -> bool:
    from slither.core.cfg.node import NodeType
    from slither.slithir.operations import LowLevelCall

    if self._is_upgradeable_proxy is None:
        self._is_upgradeable_proxy = False



        for f in self.functions:
            if f.is_fallback:
                for node in f.all_nodes():



                    if node.type == NodeType.ASSEMBLY:
                        inline_asm = node.inline_asm
                        if inline_asm:
                            if "delegatecall" in inline_asm:
                                self._is_upgradeable_proxy = True
                                return self._is_upgradeable_proxy
    return self._is_upgradeable_proxy
```

This second check never gets past ‘`if inline_asm:`’ because ‘`node.add_inline_asm()`’ never gets called by `slither.solc_parsing.declarations.function.FunctionSolc._parse_statement()`

```
def _parse_statement(
    self, statement: Dict, node: NodeSolc, scope: Union[Scope, Function]
) → NodeSolc:
    name = statement[self.get_key()]
    if name == "IfStatement":
        ...
    elif name == "InlineAssembly":
    # Added with solc 0.6 - the yul code is an AST
        if "AST" in statement and not self.compilation_unit.core.skip_assembly:
            self._function.contains_assembly = True
            yul_object = self._new_yul_block(statement["src"], scope)
            entrypoint = yul_object.entrypoint
            exitpoint = yul_object.convert(statement["AST"])
            # technically, entrypoint and exitpoint are YulNodes and we should be returning a
            # NodeSolc here but they both expose an underlying_node so oh well
            link_underlying_nodes(node, entrypoint)
            node = exitpoint
        else:
            asm_node = self._new_node(NodeType.ASSEMBLY, statement["src"], scope)
            self._function.contains_assembly = True
            # Added with solc 0.4.12
            if "operations" in statement:
                asm_node.underlying_node.add_inline_asm(statement["operations"])
            link_underlying_nodes(node, asm_node)
            node = asm_node
```

The faulty `_parse_statement()` method in `FunctionSolc`

After further testing...

- As we can see in the comments in the `_parse_statement()` method on the previous slide, we can only reach the `else:` statement in which `node.add_inline_asm()` is called when using Solidity versions < 0.6.0
- This is related to the introduction of Yul objects, which represent an assembly code block as an abstract syntax tree (AST)
- The problem isn't with Yul, but how Slither creates YulNode objects when parsing functions
- While I don't completely understand the YulNodes yet, it is clear that `add_inline_asm()` is never called on the underlying Node object when this branch is taken

After further testing...

- For Solidity versions greater than or equal to 0.4.12 but less than 0.6, the unmodified algorithm does work as intended, finding instances of `delegatecall` within `node.inline_asm`, because both the `else:` and subsequent `if:` statements are satisfied
- However, if we add the line `entrypoint.underlying_node.add_inline_asm(statement)` into `_parse_statement()` before the two lines of comments, after creating the `YulNode`, then we can modify the new `is_upgradeable_proxy()` algorithm to work with this representation of the assembly

```
def _parse_statement(
    self, statement: Dict, node: NodeSolc, scope: Union[Scope, Function]
) -> NodeSolc:
    name = statement[self.get_key()]
    if name == "IfStatement":
        ...
    elif name == "InlineAssembly":
    # Added with solc 0.6 - the yul code is an AST
        if "AST" in statement and not self.compilation_unit.core.skip_assembly:
            self._function.contains_assembly = True
            yul_object = self._new_yul_block(statement["src"], scope)
            entrypoint = yul_object.entrypoint
            exitpoint = yul_object.convert(statement["AST"])
            entrypoint.underlying_node.add_inline_asm(statement)
            # technically, entrypoint and exitpoint are YulNodes and we should be returning a
            # NodeSolc here but they both expose an underlying_node so oh well
            link_underlying_nodes(node, entrypoint)
            node = exitpoint
        else:
            asm_node = self._new_node(NodeType.ASSEMBLY, statement["src"], scope)
            self._function.contains_assembly = True
            # Added with solc 0.4.12
            if "operations" in statement:
                asm_node.underlying_node.add_inline_asm(statement["operations"])
            link_underlying_nodes(node, asm_node)
            node = asm_node
```

The patched `_parse_statement()` method, with `add_inline_asm()` highlighted

```
@property
def is_upgradeable_proxy(self) -> bool:
   if self._is_upgradeable_proxy is None:

      self._is_upgradeable_proxy = False
      is_delegating = False
      delegate_to: Variable = None

      if self.fallback_function is not None:    # new @property finds fallback
         print("\n" + self._name + " has fallback function")
         for node in self.fallback_function.all_nodes():
            print(str(node.type))
            for ir in node.irs:
               if isinstance(ir, LowLevelCall):
                  print("\nFound LowLevelCall\n")
                  if ir.function_name == "delegatecall":
                     print("\nFound delegatecall in LowLevelCall\n")
                     is_delegating = True
                     if ir.destination.is_constant:
                        self._is_upgradeable_proxy = False
                        return False
                     else:
                        delegate_to = ir.destination
            ...
```

Modified algorithm: Store `is_delegating` and `delegate_to` for later

```python
@property
def is_upgradeable_proxy(self) -> bool:
    if self._is_upgradeable_proxy is None:
        ...
        if self.fallback_function is not None:
            for node in self.fallback_function.all_nodes():
                ...
                if node.type == NodeType.ASSEMBLY:
                    if node.inline_asm:
                        if "AST" in node.inline_asm and isinstance(node.inline_asm, Dict):
                            # @webthethird: inline_asm is a Yul AST for versions ≥ 0.6.0
                            for statement in node.inline_asm["AST"]["statements"]:
                                if statement["nodeType"] == "YulExpressionStatement":
                                    statement = statement["expression"]
                                if statement["nodeType"] == "YulVariableDeclaration":
                                    statement = statement["value"]
                                if statement["nodeType"] == "YulFunctionCall":
                                    if statement["functionName"]["name"] == "delegatecall":
                                        print("\nFound delegatecall in YulFunctionCall\n")
                                        is_delegating = True
                                        dest = statement["arguments"][1]
                                        if dest["nodeType"] == "YulIdentifier":
                                            delegate_to = dest["name"]
                        ...
```

Modified algorithm: Check for `delegatecall` in inline assembly as Yul object (AST)

```
@property
def is_upgradeable_proxy(self) -> bool:
   if self._is_upgradeable_proxy is None:
      ...
      if self.fallback_function is not None:
         for node in self.fallback_function.all_nodes():
            ...
            if node.type == NodeType.ASSEMBLY:
               if node.inline_asm:
                  if "AST" in node.inline_asm and isinstance(node.inline_asm, Dict):
                  ...
                  else:
                     asm_split = node.inline_asm.split("\n")
                     for asm in asm_split:
                         if "delegatecall" in asm:
                             print("\nFound delegatecall in inline asm\n")
                             is_delegating = True
                             params = asm.split("delegatecall(")[1].split(", ")
                             dest = params[1]
                             for v in self.fallback_function.variables_read:
                                 print(str(v.expression))
                                 if v.name == dest and not v.is_constant:
                                     print("Call destination " + str(v) + " is not constant\n")
                                     delegate_to = v
```

Modified algorithm: Check for `delegatecall` in inline assembly (pre-0.6.0)

```
@property
def is_upgradeable_proxy(self) -> bool:
   if self._is_upgradeable_proxy is None:
      ...
      if self.fallback_function is not None:
         for node in self.fallback_function.all_nodes():
            ...
            elif node.type == NodeType.EXPRESSION:        # finds delegatecalls when above doesn't
               expression = node.expression
               if isinstance(expression, AssignmentOperation):
                  expression = expression.expression_right
                  print("Checking right side of assignment expression...")
               if isinstance(expression, CallExpression):
                  if "delegatecall" in str(expression.called):
                     is_delegating = True
                     print("\nFound delegatecall in expression:\n" + str(expression) + "\n")
                     dest = expression.arguments[1]
                     if isinstance(dest, Identifier):
                        var = dest.value
                        if var.is_constant:
                           self._is_upgradeable_proxy = False
                           return False
                        else:
                           print("Call destination " + str(var) + " is not constant\n")
                           delegate_to = var
```

Modified algorithm: Check for `delegatecall` in Expression Node objects - works if assembly doesn't

```
@property
def is_upgradeable_proxy(self) -> bool:
   if self._is_upgradeable_proxy is None:
      ...
      # Look for implementation setter (misses ProductProxy where Factory manages implementation)
      if is_delegating and delegate_to is not None:
         print(self.name + " is delegating to " + delegate_to.name
               + "\nLooking for setImplementation\n")
         for f in self.functions:
            # Ignore invalid functions, i.e. no name, fallback and slither generated functions
            if f.name is not None and not f.name == "fallback" and "slither" not in f.name:
               print("Checking function: " + f.name)
               for v in f.variables_written:
                  if isinstance(v, Variable):
                     print(f.name + " writes to variable: " + v.name)
                     if delegate_to.name.strip("_") in v.name:
                        print("\nImplementation set by function: " + f.name + "\n")
                        self._is_upgradeable_proxy = True
                        return self._is_upgradeable_proxy

               if f.contains_assembly:
               ...
```

Modified algorithm: Look for the function that sets the implementation to a new address

```python
@property
def is_upgradeable_proxy(self) -> bool:
    ...
    if is_delegating and delegate_to is not None:
        for f in self.functions:
            if f.name is not None and not f.name == "fallback" and "slither" not in f.name:
                ...
                if f.contains_assembly:
                    for node in f.all_nodes():
                        asm = node.inline_asm
                        if asm:   # @webthethird: inline_asm not set for version ≥ 0.6.0
                            if "sstore" in asm and delegate_to.name.strip("_").lower() in asm.lower():
                                print("\nImplementation set by function: " + f.name)
                                self._is_upgradeable_proxy = True
                                return self._is_upgradeable_proxy
                        else:
                            for e in f.all_expressions():
                                if "sstore" in str(e) and delegate_to.name.strip("_") in str(e).lower():
                                    print("\nImplementation set by function: " + f.name)
                                    print("Assembly calls sstore and includes delegate_to.name: " + str(e)
                                          + "\n")
                                    self._is_upgradeable_proxy = True
                                    return self._is_upgradeable_proxy
```

Modified algorithm: Look for the function that sets the implementation (within assembly using `sstore`)

Standard Storage Slots

- A common pattern for storing and loading the implementation address is using fixed **EIP-1967 Standard Proxy Storage Slots**. For example:

```
/**
  * @dev Storage slot with the address of the current implementation.
  * This is the keccak-256 hash of "eip1967.proxy.implementation" subtracted by 1, and
is
  * validated in the constructor.
  */
 bytes32 internal constant IMPLEMENTATION_SLOT = 0x360894a13ba1a3210667c828492db98dca3 .
. .
```

Standard Storage Slots

- A common pattern for storing and loading the implementation address is using fixed **EIP-1967 Standard Proxy Storage Slots**. For example:

```
constructor(address _logic, bytes memory _data) public payable {
   assert(IMPLEMENTATION_SLOT ==
bytes32(uint256(keccak256('eip1967.proxy.implementation')) - 1));
   _setImplementation(_logic);
   if(_data.length > 0) {
     (bool success,) = _logic.delegatecall(_data);
     require(success);
   }
 }
```

Standard Storage Slots

- A common pattern for storing and loading the implementation address is using fixed **EIP-1967 Standard Proxy Storage Slots**. For example:

```
function _implementation() override internal view returns (address impl) {
  bytes32 slot = IMPLEMENTATION_SLOT;
  assembly {
    impl := sload(slot)
  }
}
```

Standard Storage Slots

- A common pattern for storing and loading the implementation address is using fixed **EIP-1967 Standard Proxy Storage Slots**. For example:

```
function _setImplementation(address newImplementation) internal {
   require(OpenZeppelinUpgradesAddress.isContract(newImplementation), "Cannot set a
proxy
           implementation to a non-contract address");
   bytes32 slot = IMPLEMENTATION_SLOT;
   assembly {
     sstore(slot, newImplementation)
   }
 }
```

Standard Storage Slots

- There is another, very similar EIP regarding fixed proxy storage spots, called **EIP-1822: Universal Upgradeable Proxy Standard (UUPS)**
- This does not seem to be as popular as EIP-1967, but together they demonstrate how common such fixed storage slot patterns are
- Both have also been studied and written about extensively by OpenZeppelin
- Like EIP-1967, the storage slot for the logic contract's address is defined as the `keccak256` hash of a standard string, in this case, `keccak256("PROXIABLE")`
- They also define a `Proxiable` contract, to be inherited by the logic contract, and which has the function `updateCodeAddress(address newAddress)` **within the logic contract**

```
contract Proxy {
    // Code position in storage is keccak256("PROXIABLE") =
    // "0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7"
    constructor(bytes memory constructData, address contractLogic) public {
        assembly { // solium-disable-line
            sstore(0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7, contractLogic)
        }
        (bool success, bytes memory _ ) = contractLogic.delegatecall(constructData); // solium-disable-line
        require(success, "Construction failed");
     }
     function() external payable {
         assembly { // solium-disable-line
             let contractLogic := sload(0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7)
             calldatacopy(0x0, 0x0, calldatasize)
             let success := delegatecall(sub(gas, 10000), contractLogic, 0x0, calldatasize, 0, 0)
             let retSz := returndatasize
             returndatacopy(0, 0, retSz)
             switch success
             case 0 {
                 revert(0, retSz)
             }
             default {
                 return(0, retSz)
  }}}}
```

The EIP-1822 Proxy contract code

```
contract Proxiable {
    // Code position in storage is keccak256("PROXIABLE") =
"0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7"

    function updateCodeAddress(address newAddress) internal {
        require(bytes32(0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7) ==
               Proxiable(newAddress).proxiableUUID(), "Not compatible" );
        assembly { // solium-disable-line
            sstore(0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7, newAddress)
        }
    }
    function proxiableUUID() public pure returns (bytes32) {
        return 0xc5f16f0fcc639fa48a6947836d9850f504798523bf8c9a3a87d5876cf622bcf7;
    }
}

contract Owned {
    address owner;

    function setOwner(address _owner) internal { owner = _owner; }
    modifier onlyOwner() {
        require(msg.sender == owner, "Only owner is allowed to perform this action");
        _;
    }
}
```

The EIP-1822 example ERC20 token logic contract code

```
contract LibraryLockDataLayout {
  bool public initialized = false;
}

contract LibraryLock is LibraryLockDataLayout {
    // Ensures no one can manipulate the Logic Contract once it is deployed.
    // PARITY WALLET HACK PREVENTION

    modifier delegatedOnly() {
        require(initialized == true, "The library is locked. No direct 'call' is allowed");
        _;
    }
    function initialize() internal {
        initialized = true;
    }
}

contact ERC20DataLayout is LibraryLockDataLayout {
  uint256 public totalSupply;
  mapping(address⇒uint256) public tokens;
}
```

The EIP-1822 example ERC20 token logic contract code

```
contract ERC20 {
    //  ...
    function transfer(address to, uint256 amount) public {
        require(tokens[msg.sender] ⩾ amount, "Not enough funds for transfer");
        tokens[to] += amount;
        tokens[msg.sender] -= amount;
    }
}

contract MyToken is ERC20DataLayout, ERC20, Owned, Proxiable, LibraryLock {

    function constructor1(uint256 _initialSupply) public {
        totalSupply = _initialSupply;
        tokens[msg.sender] = _initialSupply;
        initialize();
        setOwner(msg.sender);
    }
    function updateCode(address newCode) public onlyOwner delegatedOnly  {
        updateCodeAddress(newCode);
    }
    function transfer(address to, uint256 amount) public delegatedOnly {
        ERC20.transfer(to, amount);
    }
}
```

The EIP-1822 example ERC20 token logic contract code

Considerations

- Because the EIP-1822 Proxy contract delegates even the **`upgradeCodeAddress()`** function to the logic contract, confirming the validity of **`is_upgradeable_proxy()`** by looking for a method like this in the Proxy contract itself - i.e. the 2nd part of the algorithm in Slither's Contract class - may likely cause false negatives
- This problem is similar to the issue with the **`ProductProxy`** and **`ProxyFactory`** contracts included in the Solidity file Put.sol, which I discussed last week
- In both cases, the code for updating the logic contract does not exist in the Proxy itself

```
interface IProxyFactory {
   function productImplementation() external view returns (address);
   function productImplementations(bytes32 name) external view returns (address);
}

/**
* @title ProductProxy
* @dev This contract implements a proxy that is deployed by ProxyFactory,
* and it's implementation is stored in factory.
*/
contract ProductProxy is Proxy {
 /**
  * @dev Storage slot with the address of the ProxyFactory.
  * This is the keccak-256 hash of "eip1967.proxy.factory" subtracted by 1, and is
  * validated in the constructor.
  */
 bytes32 internal constant FACTORY_SLOT = 0x7a45a402e4cb6e08ebc196f20f66d5d30e67285a2a8aa80503fa409e727a4af1;
 bytes32 internal constant NAME_SLOT    = 0x4cd9b827ca535ceb0880425d70eff88561ecdf04dc32fcf7ff3b15c587f8a870;
// bytes32(uint256(keccak256('eip1967.proxy.name')) - 1)
 function _name() virtual internal view returns (bytes32 name_) {
   bytes32 slot = NAME_SLOT;
   assembly {  name_ := sload(slot)  }
 }
 function _setName(bytes32 name_) internal {
   bytes32 slot = NAME_SLOT;
   assembly {  sstore(slot, name_)  }
 }
```

ProductProxy contract, which is created and maintained by a ProxyFactory

```
 function _setFactory(address newFactory) internal {
   require(OpenZeppelinUpgradesAddress.isContract(newFactory), "Cannot set a factory to a non-contract
address");

   bytes32 slot = FACTORY_SLOT;

   assembly {
     sstore(slot, newFactory)
   }
 }

 function _factory() internal view returns (address factory_) {
   bytes32 slot = FACTORY_SLOT;
   assembly {
     factory_ := sload(slot)
   }
 }

 function _implementation() virtual override internal view returns (address) {
   address factory_ = _factory();
   if(OpenZeppelinUpgradesAddress.isContract(factory_))
       return IProxyFactory(factory_).productImplementations(_name());
   else
       return address(0);
 }
}
```

ProductProxy contract, which is created and maintained by a ProxyFactory

```
/**
* @title InitializableProductProxy
* @dev Extends ProductProxy with an initializer for initializing
* factory and init data.
*/
contract InitializableProductProxy is ProductProxy {
 /**
  * @dev Contract initializer.
  * @param factory_ Address of the initial factory.
  * @param data_ Data to send as msg.data to the implementation to initialize the proxied contract.
  * It should include the signature and the parameters of the function to be called, as described in
  * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding.
  * This parameter is optional, if no data is given the initialization call will be skipped.
  */
 function __InitializableProductProxy_init(address factory_, bytes32 name_, bytes memory data_) public
payable {
   require(_factory() == address(0));
   assert(FACTORY_SLOT == bytes32(uint256(keccak256('eip1967.proxy.factory')) - 1));
   assert(NAME_SLOT    == bytes32(uint256(keccak256('eip1967.proxy.name')) - 1));
   _setFactory(factory_);
   _setName(name_);
   if(data_.length > 0) {
     (bool success,) = _implementation().delegatecall(data_);
     require(success);
   }
 }
}
```

ProductProxy contract, which is created and maintained by a ProxyFactory

```
contract Factory is Configurable, ContextUpgradeSafe, Constants {
    using SafeERC20 for IERC20;
    using SafeMath for uint;
    using SafeMath for int;

    mapping(bytes32 => address) public productImplementations;
    mapping(address => mapping(address => mapping(uint => mapping(uint => address)))) public calls;
    // _underlying => _currency => _priceFloor => _priceCap => call
    mapping(address => mapping(address => mapping(uint => mapping(uint => address)))) public puts;
    address[] public allCalls;
    address[] public allPuts;

    function length() public view returns (uint) {
        return allCalls.length;
    }

    uint public feeRate;

    function setFee(uint feeRate_, address feeTo) public governance {
        require(feeRate_ <= MAX_FEE_RATE);
        feeRate = feeRate_;
        config[_feeTo_] = uint(feeTo);
    }

    ...
```

Factory contract, which creates ProductProxy contracts, stores their implementation addresses, and more

```
contract Factory is Configurable, ContextUpgradeSafe, Constants {
    ...

    function __Factory_init(address governor, address implCall, address implPut, address WETH, address feeTo)
public initializer {
        __Governable_init_unchained(governor);
        __Factory_init_unchained(implCall, implPut, WETH, feeTo);
    }

    function __Factory_init_unchained(address implCall, address implPut, address WETH, address feeTo) public
governance {
        productImplementations[_Call_]  = implCall;
        productImplementations[_Put_]   = implPut;
        config[_WETH_]                  = uint(WETH);
        //config[_uniswapRounter_]        = uint(0x7a250d5630B4cF539739dF2C5dAcb4c659F2488D);
        setFee(0.005 ether, feeTo);     // 0.5%
    }

    function upgradeProductImplementationsTo(address implCall, address implPut) external governance {
        productImplementations[_Call_] = implCall;
        productImplementations[_Put_]  = implPut;
    }

        // it's about to get a lot more complicated...
```

Factory contract, which creates ProductProxy contracts, stores their implementation addresses, and more

```
contract Factory is Configurable, ContextUpgradeSafe, Constants {
    ...
    function createOption(address underlying, address currency, uint priceFloor, uint priceCap) public returns (address
call_, address put) {
        require(underlying ≠ currency, 'IDENTICAL_ADDRESSES');
        require(underlying ≠ address(0) && currency ≠ address(0), 'ZERO_ADDRESS');
        require(priceFloor < priceCap, 'priceCap should biger than priceFloor');
        require(config[_permissionless_] ≠ 0 || _msgSender() == governor);

        require(calls[underlying][currency][priceFloor][priceCap] == address(0), 'the Call/Put exist already');
        // single check is sufficient

        bytes memory bytecode = type(InitializableProductProxy).creationCode;

        bytes32 salt = keccak256(abi.encodePacked(_Call_, underlying, currency, priceFloor, priceCap));
        assembly {
            call_ := create2(0, add(bytecode, 32), mload(bytecode), salt)
        }
        InitializableProductProxy(payable(call_)).__InitializableProductProxy_init(address(this), _Call_,
abi.encodeWithSignature('__Call_init(address,address,uint256,uint256)', underlying, currency, priceFloor, priceCap));

        salt = keccak256(abi.encodePacked(_Put_, underlying, currency, priceFloor, priceCap));
        assembly {
            put := create2(0, add(bytecode, 32), mload(bytecode), salt)
        }
        InitializableProductProxy(payable(put)).__InitializableProductProxy_init(address(this), _Put_,
abi.encodeWithSignature('__Put_init(address,address,uint256,uint256)',  underlying, currency, priceFloor, priceCap));

        calls[underlying][currency][priceFloor][priceCap] = call_;
        puts [underlying][currency][priceFloor][priceCap] = put;
        allCalls.push(call_);
        allPuts.push(put);
        emit CreateOption(_msgSender(), underlying, currency, priceFloor, priceCap, call_, put, allCalls.length);
    }
    event CreateOption(address indexed creator, address indexed underlying, address indexed currency, uint priceFloor,
                       uint priceCap, address call, address put, uint count);
```

Edge Cases in Detecting Proxy Contracts with Slither: Part 2

Bill Bodell
August 5, 2021

A Simpler Proxy Contract

- The following proxy contract would have worked as intended in the original Slither algorithm for `is_upgradeable_proxy()`, even after removing the check for the word “proxy” in the contract’s name
- This contract was submitted for verification to Etherscan on August 28th, 2020, but it may in fact be several years older, as it specifies Solidity version 0.5.16 and does not adhere to any of the most common proxy patterns, such as those used by OpenZeppelin
- It is the first example I have found in which the `delegatecall` is found in a `LowLevelCall` SlithIR operation, rather than in an assembly expression

```
pragma solidity ^0.5.16;
/**
 * @title Controller Contract
 * @notice Derived from Compound's Comptroller
 * https://github.com/compound-finance/compound-protocol/tree/master/contracts
 */
contract ControllerAdminStorage {
    /**
    * @notice Administrator for this contract
    */
    address public admin;

    /**
    * @notice Pending administrator for this contract
    */
    address public pendingAdmin;

    /**
    * @notice Active brains of ProxyController
    */
    address public controllerImplementation;

    /**
    * @notice Pending brains of ProxyController
    */
    address public pendingControllerImplementation;
}
```

ControllerAdminStorage has simple, public state variables to store implementation and admin addresses

```
contract ProxyController is ControllerAdminStorage, ArtemcontrollerErrorReporter {

    event NewPendingImplementation(address oldPendingImplementation, address newPendingImplementation);
    event NewImplementation(address oldImplementation, address newImplementation);
    event NewPendingAdmin(address oldPendingAdmin, address newPendingAdmin);
    event NewAdmin(address oldAdmin, address newAdmin);

    constructor() public {
        // Set admin to caller
        admin = msg.sender;
    }

    /*** Admin Functions ***/
    function _setPendingImplementation(address newPendingImplementation) public returns (uint) {
        if (msg.sender != admin) {
            return fail(Error.UNAUTHORIZED, FailureInfo.SET_PENDING_IMPLEMENTATION_OWNER_CHECK);
        }
        address oldPendingImplementation = pendingControllerImplementation;
        pendingControllerImplementation = newPendingImplementation;

        emit NewPendingImplementation(oldPendingImplementation, pendingControllerImplementation);
        return uint(Error.NO_ERROR);
    }
```

The ProxyController contract uses ControllerAdminStorage rather than EIP-1967 standard storage slots

```
contract ProxyController is ControllerAdminStorage, ArtemcontrollerErrorReporter {
    ...

    /**
    * @notice Accepts new implementation of controller. msg.sender must be pendingImplementation
    * @dev Admin function for new implementation to accept it's role as implementation
    * @return uint 0=success, otherwise a failure (see ErrorReporter.sol for details)
    */
    function _acceptImplementation() public returns (uint) {
        // Check caller is pendingImplementation and pendingImplementation ≠ address(0)
        if (msg.sender != pendingControllerImplementation || pendingControllerImplementation == address(0)) {
            return fail(Error.UNAUTHORIZED, FailureInfo.ACCEPT_PENDING_IMPLEMENTATION_ADDRESS_CHECK);
        }

        // Save current values for inclusion in log
        address oldImplementation = controllerImplementation;
        address oldPendingImplementation = pendingControllerImplementation;

        controllerImplementation = pendingControllerImplementation;
        pendingControllerImplementation = address(0);

        emit NewImplementation(oldImplementation, controllerImplementation);
        emit NewPendingImplementation(oldPendingImplementation, pendingControllerImplementation);

        return uint(Error.NO_ERROR);
    }
```

The ProxyController contract uses ControllerAdminStorage rather than EIP-1967 standard storage slots

```
contract ProxyController is ControllerAdminStorage, ArtemcontrollerErrorReporter {
    ...

    /**
     * @dev Delegates execution to an implementation contract.
     * It returns to the external caller whatever the implementation returns
     * or forwards reverts.
     */
    function () payable external {
        // delegate all other functions to current implementation
        (bool success, ) = controllerImplementation.delegatecall(msg.data);

        assembly {
              let free_mem_ptr := mload(0x40)
              returndatacopy(free_mem_ptr, 0, returndatasize)

              switch success
              case 0 { revert(free_mem_ptr, returndatasize) }
              default { return(free_mem_ptr, returndatasize) }
        }
    }
```

The ProxyController contract uses ControllerAdminStorage rather than EIP-1967 standard storage slots

```
def is_upgradeable_proxy(self) -> bool:
    from slither.core.cfg.node import NodeType
    from slither.slithir.operations import LowLevelCall

    if self._is_upgradeable_proxy is None:
        self._is_upgradeable_proxy = False
        if "Proxy" in self.name:
            self._is_upgradeable_proxy = True
            return True
        for f in self.functions:
            if f.is_fallback:
                for node in f.all_nodes():
                    for ir in node.irs:
                        if isinstance(ir, LowLevelCall) and ir.function_name == "delegatecall":
                            self._is_upgradeable_proxy = True
                            return self._is_upgradeable_proxy
                    if node.type == NodeType.ASSEMBLY:
                         ...
```

Recall the 1st part of the original `is_upgradeable_proxy()` algorithm, which never returns true with EIP-1967

```
def is_upgradeable_proxy(self) -> bool:

   if self._is_upgradeable_proxy is None:
       self._is_upgradeable_proxy = False
       is_delegating = False
       delegate_to: Variable = None
       if self.fallback_function is not None:
           print("\n" + self._name + " has fallback function\n")
           for node in self.fallback_function.all_nodes():
               print(str(node.type))
               for ir in node.irs:
                   if isinstance(ir, LowLevelCall):
                       print("\nFound LowLevelCall\n")
                       if ir.function_name == "delegatecall":
                           print("\nFound delegatecall in LowLevelCall\n")
                           is_delegating = True
                           if ir.destination.is_constant:
                               self._is_upgradeable_proxy = False
                               return False
                           else:
                               delegate_to = ir.destination
                               print("Call destination " + str(delegate_to) + " is not constant\n")
                               break
               if is_delegating and delegate_to is not None:
                   break
               if node.type == NodeType.ASSEMBLY:
                    ...
```

The same portion of our modified algorithm, after testing using this contract

More Complex Proxy Contracts (based on EIP-1967)

- As seen before in the case of the ProductProxy and Factory contracts, many extensions of EIP-1967 involve proxies that are created and managed by a variety of other contract types
- The addresses of these contracts are stored in the same way as the implementation address is, using standard memory slots determined by computing the following:
  `keccak256("eip1967.proxy.[contract-type]") - 1`
  where `contract-type` may be replaced with 'admin', 'factory', 'beacon', etc.
- We saw a similar, albeit seemingly much less popular standard in EIP-1822
- However, EIP-1967 seems to be preferred for its extendability, i.e. by using informative dot notation in the hashed strings

More Complex Proxy Contracts (based on EIP-1967)

- We have seen that our algorithm is currently unable to determine whether such a managed proxy contract is in fact upgradeable because it cannot find an implementation setter method in the proxy contract itself
- In such cases, if we check the implementation getter, we should be able to find a constant variable with ‘slot’ in the name, and use that to determine what type of contract the proxy relies on to retrieve the implementation address, i.e. a proxy factory, admin, beacon, etc.
- We should also be able to discover contracts in the wild which appear to comply with EIP-1967 yet break with the standard format of the string that is hashed

```
/**
  * @dev Storage slot with the address of the ProxyFactory.
  * This is the keccak-256 hash of "eip1967.proxy.factory" subtracted by 1, and is
  * validated in the constructor.
  */
 bytes32 internal constant FACTORY_SLOT = 0x7a45a402e4cb6e08ebc196f20f66d5d30e67285a2a8aa80503fa409e727a4af1;
// bytes32(uint256(keccak256('eip1967.proxy.name')) - 1)
 bytes32 internal constant NAME_SLOT    = 0x4cd9b827ca535ceb0880425d70eff88561ecdf04dc32fcf7ff3b15c587f8a870;

 function _name() virtual internal view returns (bytes32 name_) {
   bytes32 slot = NAME_SLOT;
   assembly {  name_ := sload(slot)  }
 }

 function _factory() internal view returns (address factory_) {
   bytes32 slot = FACTORY_SLOT;
   assembly {  factory_ := sload(slot)  }
 }

 function _implementation() virtual override internal view returns (address) {
   address factory_ = _factory();
   if(OpenZeppelinUpgradesAddress.isContract(factory ))
       return IProxyFactory(factory_).productImplementations(_name());
   else
       return address(0);
 }
```

From ProductProxy, the implementation getter uses a proxy factory interface and two EIP-1967 slots

```
interface IProxyFactory {
   function productImplementation() external view returns (address);
   function productImplementations(bytes32 name) external view returns (address);
}

contract Factory is Configurable, ContextUpgradeSafe, Constants {
   using SafeERC20 for IERC20;
   using SafeMath for uint;
   using SafeMath for int;

   mapping(bytes32 => address) public productImplementations;
   ...

   function upgradeProductImplementationsTo(address implCall, address implPut) external governance {
       productImplementations[_Call_] = implCall;
       productImplementations[_Put_]  = implPut;
   }
}
```

Knowing what to look for, we should be able to find this Factory contract in the SlitherCompilationUnit

```
/**
* @dev Implements a proxy that gets the implementation address for each call from a {UpgradeableBeacon}.
* The beacon address is stored in storage slot `uint256(keccak256('eip1967.proxy.beacon')) - 1`, so that it
* doesn't conflict with the storage layout of the implementation behind the proxy.
*/
contract BeaconProxy is Proxy {
    /**
     * @dev Storage slot of the UpgradeableBeacon contract which defines the implementation for this proxy.
     * This is bytes32(uint256(keccak256('eip1967.proxy.beacon')) - 1)) and is validated in the constructor.
     */
    bytes32 private constant BEACON_SLOT = 0xa3f0ad74e5423aebfd80d3ef4346578335a9a72aeaee59ff6cb3582b35133d50;

    /**
     * @dev Initializes the proxy with `beacon`.
     *
     * If `data` is nonempty, it's used as data in a delegate call to the implementation returned by the
     * beacon. This will typically be an encoded function call, and allows initializating the storage of the
     * proxy like a Solidity constructor.
     *
     * Requirements:
     * - `beacon` must be a contract with the interface {IBeacon}.
     */
    constructor(address beacon, bytes memory data) public payable {
        assert(BEACON_SLOT == bytes32(uint256(keccak256("eip1967.proxy.beacon")) - 1));
        _setBeacon(beacon, data);
    }
```

Another example where finding the constant slot variable and looking for an interface in the getter can help us find the accompanying beacon contract which stores the implementation address

```
/**
* @dev Implements a proxy that gets the implementation address for each call from a {UpgradeableBeacon}.
* The beacon address is stored in storage slot `uint256(keccak256('eip1967.proxy.beacon')) - 1`, so that it
* doesn't conflict with the storage layout of the implementation behind the proxy.
*/
contract BeaconProxy is Proxy {

    ...

    /**
     * @dev Returns the current beacon address.
     */
    function _beacon() internal view virtual returns (address beacon) {
        bytes32 slot = _BEACON_SLOT;
        // solhint-disable-next-line no-inline-assembly
        assembly {
            beacon := sload(slot)
        }
    }

    /**
     * @dev Returns the current implementation address of the associated beacon.
     */
    function _implementation() internal view virtual override returns (address) {
        return IBeacon(_beacon()).implementation();
    }
```

Another example where finding the constant slot variable and looking for an interface in the getter can help us find the accompanying beacon contract which stores the implementation address

```
/**
* @dev Implements a proxy that gets the implementation address for each call from a {UpgradeableBeacon}.
* The beacon address is stored in storage slot `uint256(keccak256('eip1967.proxy.beacon')) - 1`, so that it
* doesn't conflict with the storage layout of the implementation behind the proxy.
*/
contract BeaconProxy is Proxy {
    ...
    /**
     * @dev Changes the proxy to use a new beacon. If `data` is nonempty, it's used as data in a delegate call
     * to the implementation returned by the beacon.
     * Requirements:
     * - `beacon` must be a contract.
     * - The implementation returned by `beacon` must be a contract.
     */
    function _setBeacon(address beacon, bytes memory data) internal virtual {
        require(Address.isContract(beacon), "BeaconProxy: beacon is not a contract");
        require(Address.isContract(IBeacon(beacon).implementation()),
                "BeaconProxy: beacon implementation is not a contract");
        bytes32 slot = _BEACON_SLOT;
        assembly {
            sstore(slot, beacon)
        }
        if (data.length > 0) {
            Address.functionDelegateCall(_implementation(), data, "BeaconProxy: function call failed");
        }
    }
}
```

Another example where finding the constant slot variable and looking for an interface in the getter can help us find the accompanying beacon contract which stores the implementation address

```
/**
* @dev This contract is used in conjunction with one or more instances of {BeaconProxy} to determine their
* implementation contract, which is where they will delegate all function calls. An owner is able to change
* the implementation the beacon points to, thus upgrading the proxies that use this beacon.
*/
contract UpgradeableBeacon is IBeacon, Ownable {
   address private _implementation;

   /**
    * @dev Emitted when the implementation returned by the beacon is changed.
    */
   event Upgraded(address indexed implementation);

   /**
    * @dev Sets the address of the initial implementation, and the deployer account as the owner who can
    * upgrade the beacon.
    */
   constructor(address implementation_) public {
       _setImplementation(implementation_);
   }

   /**
    * @dev Returns the current implementation address.
    */
   function implementation() public view virtual override returns (address) {
       return _implementation;
   }
```

Another example where finding the constant slot variable and looking for an interface in the getter can help us find the accompanying beacon contract which stores the implementation address

```
/**
* @dev This contract is used in conjunction with one or more instances of {BeaconProxy} to determine their
* implementation contract, which is where they will delegate all function calls. An owner is able to change
* the implementation the beacon points to, thus upgrading the proxies that use this beacon.
*/
contract UpgradeableBeacon is IBeacon, Ownable {
    ...
    /**
     * @dev Upgrades the beacon to a new implementation. Emits an {Upgraded} event.
     * Requirements:
     * - msg.sender must be the owner of the contract.
     * - `newImplementation` must be a contract.
     */
    function upgradeTo(address newImplementation) public virtual onlyOwner {
        _setImplementation(newImplementation);
        emit Upgraded(newImplementation);
    }

    /**
     * @dev Sets the implementation contract address for this beacon
     * Requirements:
     * - `newImplementation` must be a contract.
     */
    function _setImplementation(address newImplementation) private {
        require(Address.isContract(newImplementation), "UpgradeableBeacon: implementation is not a contract");
        _implementation = newImplementation;
    }
}
```

Another example where finding the constant slot variable and looking for an interface in the getter can help us find the accompanying beacon contract which stores the implementation address

```
contract BaseAdminUpgradeabilityProxy is BaseUpgradeabilityProxy {
   ...
   /**
    * @dev Storage slot with the address of the current implementation.
    * This is the keccak-256 hash of "bts.lab.eth.proxy.impl", and is
    * validated in the constructor.
    */
   bytes32 internal constant IMPLEMENTATION_SLOT =
0xe99d12b39ab17aef0ca754554afa48519dcb96ca64603696637dea37e965a617;

   /**
    * @dev Storage slot with the admin of the contract.
    * This is the keccak-256 hash of "bts.lab.eth.proxy.admin", and is
    * validated in the constructor.
    */
   bytes32 internal constant ADMIN_SLOT =
0xd605002b0407d620d5ea33643507867180e600a98b93d382fc50227c2095905e;
```

In this example, the rest of the source code appears identical to other contracts by the same name, but the slots do not comply with EIP-1967 because the strings they are derived from have been changed

```
/**
 * @title Proxy
 * @dev Implements delegation of calls to other contracts, with proper
 * forwarding of return values and bubbling of failures.
 * It defines a fallback function that delegates all calls to the address
 * returned by the abstract _implementation() internal function.
 */
contract SlaveProxy {

 bytes32 private constant MANAGER_SLOT = 0x7a55c4d64d3f68c3935ebba18bdf734d8a1d1d068c865f9e08eab9d3a6da73b4;

 /**
  * @dev Contract constructor.
  * @param manager Address of the proxy manager.
  * @param data Data to send as msg.data to the implementation to initialize the proxied contract.
  * It should include the signature and the parameters of the function to be called, as described in
  * https://solidity.readthedocs.io/en/v0.4.24/abi-spec.html#function-selector-and-argument-encoding.
  * This parameter is optional, if no data is given the initialization call will be skipped.
  */
 constructor(address manager, bytes data) public {
   assert(MANAGER_SLOT == keccak256("minuteman-wallet-manager"));
   setManager(manager);

   if(data.length > 0) {
     require(_implementation().delegatecall(data));
   }
 }
```

One more example which deviates from the EIP-1967 standard, but only because of the hashed string

```
/**
 * @title Proxy
 * @dev Implements delegation of calls to other contracts, with proper
 * forwarding of return values and bubbling of failures.
 * It defines a fallback function that delegates all calls to the address
 * returned by the abstract _implementation() internal function.
 */
contract SlaveProxy {
   ...

 function _implementation() internal view returns (address) {
   return WalletManager(managerAddress()).getImplementation();
 }

 function setManager(address manager) internal {
   bytes32 slot = MANAGER_SLOT;
   assembly {
     sstore(slot, manager)
   }
 }

 function managerAddress() internal view returns(address manager) {
   bytes32 slot = MANAGER_SLOT;
   assembly {
     manager := sload(slot)
   }
 }
```

One more example which deviates from the EIP-1967 standard, but only because of the hashed string

```
contract WalletManager is Initializable, Ownable {

 mapping(address => address) public walletsByUser;
 address private implementation;

 event UserWalletCreated(address user, address walletAddress);
 event ImplementationChanged(address implementation);

 function initialize(address _implementation) initializer public {
   Ownable.initialize(msg.sender);
   implementation =  implementation;
   emit ImplementationChanged(implementation);
 }

 function getImplementation() external view returns (address) {
   return implementation;
 }

 function setImplementation(address newImplementation) external onlyOwner {
   implementation = newImplementation;
   emit ImplementationChanged(implementation);
 }
```

One more example which deviates from the EIP-1967 standard, but only because of the hashed string

```
contract WalletManager is Initializable, Ownable {
 ...
 function createWallet(address owner) public returns (address) {
   require(owner == address(0x0) || walletsByUser[owner] == address(0x0),
           "Address already has existing wallet");

   bytes memory data = abi.encodeWithSignature("initialize(address,address)",
                                               address(this), owner);
   address proxy = new SlaveProxy(address(this), data);

   if (owner != address(0x0)) {
     walletsByUser[owner] = proxy;
   }
   emit UserWalletCreated(owner, proxy);
   return proxy;
 }

 function changeOwner(address oldOwner, address newOwner) public {
   require(oldOwner == address(0) || msg.sender == walletsByUser[oldOwner]);
   walletsByUser[oldOwner] = address(0);
   walletsByUser[newOwner] = msg.sender;
 }
}
```

One more example which deviates from the EIP-1967 standard, but only because of the hashed string