CERTORA

Formal Verification of Furucombo

Summary

This document describes the specification and verification of Furucombo using Certora Prover. The work was undertaken from March 15, 2021 - April 12, 2021. The latest commit that was reviewed and run through the Certora Prover was c90dec1046e5e26edf70f985dd86381403912482.

The scope of our verification was the Proxy, Registry, and Handlers functionality, with focus on permissions, state integrity, avoiding execution of malicious external code, and valid token movements. The handlers in scope were: HMaker, HAaveProtocol, HAaveProtocolV2, HSCompound, HFunds, HOneInchExchange, HBalancer, HCEther, HCToken, HGasTokens,HStakingRewardsAdapter, HUniswapV2, HCurve, HCurveDao, HWeth, HYVault, HKyberNetwork, HSushiSwap, HUniswap, HMooniswap, HFurucomboStaking, HComptroller, HBProtocol. The parts of the inline assembly that are not memory safe were simplified to allow high level reasoning about the flow and contract integrations. Properties of the underlying DeFi protocols were assumed but not verified.

The Certora Prover proved the implementations of the Proxy, Registry and the listed handlers are correct with respect to the formal rules written by the Furucombo and the Certora teams. During the verification process, the Certora Prover discovered minor bugs in the code listed in the table below. All issues were promptly corrected, and the fixes were verified to satisfy the specifications up to the limitations of the Certora Prover. The Certora development team is currently handling these limitations. The next section formally defines high level specifications of the contracts in scope. **All the rules are publically available in a public github https://github.com/dinngodev/furucombo-contract.**

During the project we found 1 high severity issue, 6 medium severity issues, 7 low severity issues, and 2 recommendations.

List of Main Issues Discovered

Severity: High

Issue:	Potential memory overwrite of the local stack
Rules Broken:	**Memory safety**
Description:	In proxy.sol:_parse, if the handler returns length in [1..31], the new index will remain equal to index in _parse, thus allowing to overwrite it in the next iteration of the loop in _execs.
Fix:	The expectations about valid handler outputs should be checked: the return length is checked to be a multiple of 32.

Severity: Medium

Issue:	HAaveProtocol executeOperation can be delegatecall-ed directly from Proxy
Rules Broken:	**onlyValidCaller; The *executeOperation* should only be *delegatecalled* by the *Proxy* when the *flashLoan* is initiated by Furucombo.**
Description:	The executeOperation serves as the callback for the flashloan, but in fact it can be called directly from the proxy too. This can be confusing to users and may lead to donation of funds to the lending pool.
Fix:	executeOperation should be called by a valid caller, in particular the Aave contract.

Severity: Medium

Issue:	Handler outputs are limited to types that are in multiples of 32 bytes, otherwise return data length and number of return values may not match
Rules Broken:	**Memory safety**

Description:	In Proxy.sol:_execs, if the handler returns a buffer of length in [1..31], and the number of arguments is 0, then data is copied into the local stack while not matching the 32-byte offset packing. The require may pass trivially if the config is set to match to the newly computed newIndex.
Fix:	Handler return buffers should be 32-byte aligned to be parsed correctly - fixed.

Severity: Medium

Issue:	Untrusted handler may overwrite memory
Rules Broken:	**Memory safety**
Description:	In Proxy.sol:_exec - returndatasize can be arbitrary, and cause a memory overflow when computing the new value of free memory pointer in _exec. For comparison, when the Solidity compiler handles a function that returns a dynamic type parameter such as an array, it adds various checks that bound the length of the array returned, and forces the increase to be 32-byte aligned.
Mitigation:	Overflow of the free memory pointer due to huge returndatasize value is infeasible thanks to gas.

Severity: Medium

Issue:	Getter functions should be marked view
Rules Broken:	**noOverwrite**
Description:	Calling external contracts may lead to reentrancy which in turn could have adverse effects, such as modifying the state of the contract in unexpected ways. For example: - IAToken underlyingAssetAddress - IComptroller getCompAddress
Fix:	External functions that are not expected to modify the state should be marked view in the relevant interface, to guarantee that the

	Solidity compiler uses STATICCALL opcode which is safer reentrancy-wise.

Severity: Medium

Issue:	batchExec can be called if sender is already set.
Rules Broken:	**Only a limited number of operations are allowed when the sender is initialized**
Description:	batchExec unexpectedly succeeds despite the sender being initialized. This is because setSender() is idempotent and does not revert if trying to run it a second time (see related issue). This may mean reentrancy is allowed and hijacking of funds is possible. The other checks (that the cube counter is zero, and that the stack is empty) should make such a reentrancy impossible, but reasoning about it is more complicated and thus could be prone to errors in future versions. (For example, if an attacker is able to overflow the stack length field, and the cube counter, then such a reentrancy is made possible.)
Fix:	halt and banned agent checks are added before external functions in Proxy.

Severity: Medium

Issue:	HFunds: getBalance should have the same conditions as _getBalance
Description:	The _getBalance function in handler base is using 0xeee.. and 0x0 as both indicators to using native ETH token. But HFund's getBalance function is only comparing against 0x0. This is confusing and potentially could cause errors in handlers' executions.
Fix:	HFunds getBalance was changed to call to the internal _getBalance function.

Severity: Low

Issue:	HFunds: sendTokens can have unexpected eth balance modification

Description:	The token “0” address can be passed multiple times, and nowhere it is checked that the sum of eth amounts is equal to msg.value (or to the current working balance).

Severity: Low

Issue:	Can unregister a caller or handler via the matching register function
Rules Broken:	**Registration of handlers can only be modified by the relevant functions**
Description:	Registry.sol - One can deprecate a handler via the register function, or a caller via the registerCaller function, by giving the “deprecated” info directly. This is potentially confusing since a dedicated function exists for that end.
Fix:	Disallow deprecation via register functions.

Severity: Low

Issue:	2 issues: Halting can be irreversible; banning can be irreversible
Rules Broken:	**haltingIsReversible, banningIsReversible**
Description:	Registry.sol - If ownership is renounced in a state where the system is halted, it will be impossible to go back to non halted state. Same is true for the state where an agent is banned, and ownership is renounced. The agent can never be unbanned.
Mitigation:	No action necessary.

Severity: Low

Issue:	batchExec can be called on a proxy that was banned by its registry
Rules Broken:	**nonExecutableByBannedAgents**
Description:	Proxy.sol/Registry.sol - A proxy is also called an agent by the registry. Banning the proxy in the registry disallows certain callbacks

	to be calling into the proxy, but the top-level batchExec can still be called, and even fully execute (unless there are callbacks to it).
Fix:	Extend the check of banned agent + halting to batchExec.

Severity: Low

Issue:	Can set a sender a second time without any update
Rules Broken:	**nonExecutableWithInitializedSender**
Description:	Storage.sol - The_setSender() function can be called more than once with a non-zero address, but the second call has no effect. This could lead to unexpected results. (This issue is related to "batchExec can be called if sender is already set." which is the more concrete effect of this behavior).
Fix:	Revert in _setSender() if the sender is already set.

Severity: Low

Issue:	In LibStack.sol:setHandlerType, _input can have type HandlerType instead of uint256
Description:	enum type are uint8 and have Solidity compiler generated checks of compliance, so it's recommended to use. This also relieves of the need to check the input against uint96.
Fix:	Change the type of _input to the enum type

Severity: Low

Issue:	Unused return variable
Description:	In Registry.sol and Proxy.sol, the functions isValidHandler(), isValidCaller(), _isValidHandler() and _isValidCaller(), return \`result\` by their definition, although this variable's value is never set.
Fix:	Omit the return variable name from the function definition.

Severity: Recommendation

Issue:	Hard to read code
Description:	In Proxy.sol, in _execs() function, the code block handling referenced configs can be simplified by hoisting the call to _exec. In LibParam.sol, the loop counting references in getParams() function can be simplified.

Severity: Recommendation

Issue:	Use an immutable for the registry address
Description:	Storage.sol/Proxy.sol - In case a vulnerable or malicious handler is executed via delegatecall, one privilege escalation method is to override the value stored at the registry slot to point to an alternative registry that allows malicious handlers and callers. Solidity's immutable feature allows to hard-code the address of the proxy's registry.

CERTORA

Disclaimer

The Certora Prover takes as input a contract and a specification and formally proves that the contract satisfies the specification in all scenarios. Importantly, the guarantees of the Certora Prover are scoped to the provided specification, and any cases not covered by the specification are not checked by the Certora Prover.

We hope that this information is useful, but provide no warranty of any kind, express or implied. The contents of this report should not be construed as a complete guarantee that the contract is secure in all dimensions. In no event shall Certora or any of its employees be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from, out of or in connection with the results reported here.

Notations

1. ✔️ indicates the rule is formally verified on the latest commit. We write ✔️* when the rule was verified on a simplified version of the code (or under some assumptions).
2. ❌ indicates the rule was violated in a test version of the code.
3. 🔎 indicates that details per function are given below the rule formulation.
4. ✍ indicates the rule is not yet formally specified.
5. 🔁 indicates the rule is postponed **(<due to other issue, low priority?>)** .
6. We use Hoare triples of the form {p} C {q}, which means that if the execution of program C starts in any state satisfying p, it will end in a state satisfying q. In Solidity, p is similar to require, and q is similar to assert.
   The syntax {p} ($C_1 \sim C_2$) {q} is a generalization of Hoare rules, called relational properties. {p} is a requirement on the states before C_1 and C_2, and {q} describes the states after their executions. Notice that C_1 and C_2 result in different states. As a special case, $C_1 \sim_{op} C_2$, where op is a getter, indicating that C_1 and C_2 result in states with the same value for op.

Verification of Registry

The Registry is responsible for access control of the Proxy instances. In particular, it stores information about registered handlers and callers, deprecated handlers and callers, and handlers associated with callers. It can also control which Proxy instances can interact with it, and allow to pause the system.

Functions

owner() : address
Returns the owner, who may execute the sensitive functions exposed by the Registry.

handlers(address handler) : bytes32
Returns information about a registered handler, or 0 if it is unregistered.

callers(address caller) : bytes32
Returns information about a registered caller, or 0 if it is unregistered.

register(address handler, bytes32 info)
Registers the handler with the provided information.

registerCaller(address caller, bytes32 info)
Registers the caller with the provided information.

unregister(address handler)
Deprecates the handler.

unregisterCaller(address caller)
Deprecates the caller.

banned(address agent) : boolean
Returns true if the agent is banned.

ban(address agent)
Marks the agent as banned.

unban(address agent)
Marks the agent as unbanned.

isHalted(): boolean
Returns true if the system is halted.

halt()
Marks the system as halted.

unhalt()
Marks the system as unhalted.

deprecated() : bytes32
Returns the deprecated value for a handler or caller. Once a handler or a caller were deprecated, they cannot be reinstated as valid.

Properties

1. **Privileged operations ✔**
   Registration of handlers and callers, banning of agents, and halting can be executed by the owner only.

2. **Registration of handlers can only be modified by the relevant functions (deprecatesHandler, changesHandler) ❌**

```
{ before = handlers(handler) }
      op()
{ after = handlers(handler) ∧
   ((op ≠ register ∧ op ≠ unregister) ⇒ after = before) ∧
   (op ≠ unregister) ⇒ (before ≠ deprecated() ⇒ after ≠ deprecated())
}
```

3. **Unregistration of handlers cannot be undone in a single step ✔**

```
{ }
      unregisterHandler(handler)
      op()
{ handlers(handler) = deprecated() }
```

4. **Unregistration of handlers cannot be undone (cannot change deprecated state)** ✔

```
{ handlers(handler) = deprecated() }
      op()
{ handlers(handler) = deprecated() }
```

5. **Registration of callers can only be modified by the relevant functions (deprecatesCaller, changesCaller)** ❌

```
{ before = callers(caller) }
             op()
{ after = callers(caller) ∧
   ((op ≠ registerCaller ∧ op ≠ unregisterCaller) ⇒ after = before)
∧
   (op ≠ unregisterCaller) ⇒
      (before ≠ deprecated() ⇒ after ≠ deprecated())
}
```

6. **Unregistration of callers cannot be undone in a single step** ✔

```
{ }
      unregisterHandler(handler);
      op()
{ handlers(handler) = deprecated() }
```

7. **Unregistration of callers cannot be undone (cannot change deprecated state)** ✔

```
{ callers(caller) = deprecated() }
      op()
{ callers(caller) = deprecated() }
```

8. **Banning is reversible** ❌

```
{ owner() ≠ 0 ∧ op ≠ unban }
      ban(agent) by someSender;
      op();
      success = unban(agent) by someSender;
{ success }
```

CERTORA

9. Halting is reversible ❌

```
{ owner() ≠ 0 ∧ op ≠ unhalt }
      halt() by someSender;
      op();
      success = unhalt() by someSender;
{ success }
```

Verification of Proxy

The Proxy is the main entry point into Furucombo's contracts. It is responsible for instrumenting and executing a sequence of "combo"s (handlers) that interact with the different DeFi protocols. It transfers funds and tokens to the necessary contracts. Any remaining funds should be sweeped up in the end to restore its original state. It is almost entirely stateless, and some state is stored temporarily at the transaction level. The properties below are mostly relevant to a Proxy instance but can be checked for the handlers as well.

Functions

batchExec(address[] targets, bytes32[] configs, bytes[] datas)
Main handler (combo) execution function, that can be run as a top-level transaction.

execs(address[] targets, bytes32[] configs, bytes[] datas)
Handler entry point for calling back the proxy.

fallback() / receive()
If given payload, the fallback will execute the handler matching for the (pre-allowed) caller with the given payload (by using delegatecall) and return the result.
If no payload is given, it will accept any funds that are sent, if the caller is a contract.

sender() : address
Returns the set address of the original executor of the proxy in this transaction.

sender_key() : bytes32
Returns the cache key for the sender.

cubeCounter() : uint
Returns the number of cubes/handlers invoked in the current transaction (starting from batchExec).

cubeCounter_key() : bytes32
Returns the cache key for the cube counter.

stackLength() : uint
Returns the current length of the stack, that is used for various handler cleanup processes after execution of all handlers is completed.

stackLength_slot() : uint
Returns the slot number holding the stack length.

cache(bytes32 key) : bytes32
Returns the value of the key stored in cache. The cache is used for transaction-level state storage.

getSlot(uint slot) : uint
Returns the value of the flat storage slot at the provided slot. Note that this refers only to primitive fields and not to mappings or arrays.

isHandler() : bool
Always false for the proxy. For handlers code, it will return true. (This is an important distinction since handlers' code is running in the context of the proxy's state by use of delegatecall).

Properties

10. Elements in local stack cannot be overwritten once they are set ❌

See issues list. This is a result of the assembly code not being memory-safe, thus extra care must be taken to avoid unintended functionality due to user input and bad return values by handlers.

11. Only a limited number of operations are allowed to be executed by banned agents ❌

```
{ ¬isHandler() ∧ ¬op.isView ∧ registry().banned(caller) }
      success = op() by caller
{ ¬success }
```

This rule is violated only for the batchExec function, when it is given 0 actions.

12. Only a limited number of operations are allowed when the system is halted 🔎

```
{ ¬isHandler() ∧ ¬op.isView ∧ registry().isHalted() }
      success = op()
{ ¬success }
```

This rule is violated only for thereceive function, and is intended functionality.

13. **Only a limited number of operations are allowed when the sender is uninitialized 🔎**

    ```
    { ¬isHandler() ⋀ ¬op.isView ⋀ sender() = 0 }
          success = op()
    { ¬success ⋁ op = batchExec }
    ```

    This rule is violated only for thereceive function, and is intended functionality.

14. **Only a limited number of operations are allowed when the sender is initialized ❌**

    ```
    { ¬isHandler() ⋀ ¬op.isView ⋀ sender() ≠ 0 }
          success = op()
    { ¬success }
    ```

    This rule is violated for batchExec, unexpectedly succeeding despite the sender being initialized. This is because setSender() is idempotent and does not revert if trying to run it a second time. This may mean reentrancy is allowed and allow hijacking funds. The other checks should make reentrancy impossible, but reasoning about it is complicated and thus could be prone to errors in future versions.
    The rule is also violated for execs() and the fallback function, which is intended functionality.

15. **Cube counter is greater than 0 if and only if the sender is initialized ✔**

    ```
          sender() ≠ 0 ⇔ cubeCounter() > 0
    ```

16. **[Handler-applicable-1] Stack length increase is bounded ✔**

Suppose that gas-wise the number of handlers can be run is `BOUND`, then the number of elements in the stack cannot increase by more than `BOUND`. This implies the stack cannot overflow.

```
{ len = stackLength() }
      op()
{ stackLength() ≤ len + BOUND }
```

17. [Handler-applicable-2] Sender is always cleaned up ✔

```
{ _sender = sender() }
      op()
{ _sender = 0 ⇒ sender() = 0 }
```

18. [Handler-applicable-3] Cube counter is always cleaned up 🔎

```
{ cubeCounter = cubeCounter() }
      op()
{ cubeCounter = 0 ⇒ cubeCounter() = 0 }
```

This rule expectedly fails on execs() which can be run as part of a callback.

19. [Handler-applicable-4] Cache is always cleaned up ✔

Keys that are not the sender nor the cube counter are always cleaned up from the cache after an execution.

```
{ value = cache(key) ∧ key ≠ sender_key() ∧ key ≠ cubeCounter_key() }
      op()
{ value = 0 ⇒ cache(key) = 0 }
```

20. [Handler-applicable-5] “Flat” slots are always cleaned up ✔

Slots in storage that are not map/array accesses, nor the stack length, are always cleaned up if written to, with the exception of the stack length slot.

```
{ value = getSlot(slot) }
      op()
{ value = 0 ⇒ (slot = stackLengthSlot() ∨ getSlot(slot) = 0) }
```

21. **[Handler-applicable-6] “Flat” slots that are written to are not overwritten ✔**

For slots in storage that are not map/array accesses, nor the stack length, if they are not 0 at the beginning of a transaction, they retain the same value at the end of the transaction.

```
{ value = getSlot(slot) }
      op()
{ value = getSlot(slot) ⋁ slot = stackLengthSlot() }
```

Verification of Handlers

Handlers are plug-ins that allow interaction with different DeFi protocols. They are delegatecall-ed by the proxy, meaning that they share state with the Proxy. As such, it is absolutely necessary to have in place certain guarantees about what parts of the state the handlers can modify. The below properties can be checked on both any Proxy instance as well as any handler instance. Every time a new handler is added, it should satisfy the below properties. In addition, the system must be properly configured to disallow any external code from being delegatecall-ed as a handler.

(Properties are listed in the next page.)

Properties

22. If handler sends tokens to Proxy, a post-process hook is added 🔎

Handler	Violating functions
HMaker	Due to DSProxy overapproximation - safeLockGem, safeLockETH, wipe, wipeAll, freeETH
HAaveProtocol	redeem (overapproximation - only if we redeem aETH to ETH we do not to updated the stack) flashLoan (Expected)
HAaveProtocolV2	flashloan (depending on modes - non zero mode will have tokens left, stack should be updated)
HSCompound	Due to DSProxy overapproximation: borrow, withdraw, repayBorrow (should send directly back to user) enterMarket, enterMarkets, exitMarket (should not move tokens)
HFunds	None
HOneInchExchange	None
HBProtocol	Due to DSProxy overapproximation - safeLockGem, safeLockETH, wipe, wipeAll, freeETH
HBalancer	None
HCEther	Expected in cETH: redeem, redeemUnderlying (redemption is of ETH)
HCToken	None
HComptroller	claimComp - when the holder/user is the Proxy itself - considered expected case
HCurve	Potential mismatch between index in pool and token address: exchange, removeLiquidity* (addressed)
HCurveDao	None
HFurucomboStaking	Expected due to overapproximation of claimWeeks: claimAll
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None

HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None

23. **Token approval from the proxy to the DeFi protocol is temporary** 🔎

The allowance from the Proxy contract to the DeFi protocol is a temporary one, and should be revoked after the handler call is done. This rule assumes a single handler invocation will transfer one type of token from the proxy to the protocol.

```
{ allowance = someToken.allowance(proxy, someDeFiProtocol) }
        op()
{ allowance = 0 ⇒ someToken.allowance(proxy, someDeFiProtocol) = 0 }
```

Handler	Violating functions
HMaker	None
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	None
HFunds	None
HOneInchExchange	None
HBProtocol	None
HBalancer	None
HCEther	None
HCToken	None
HComptroller	None
HCurve	exchange*, addLiquidity* removeLiquidityOneCoinDust (expected)
HCurveDao	deposit
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	addLiquidity, addLiquidityETH, removeLiquidity, removeLiquidityETH
HUniswap	None
HUniswapV2	addLiquidity, addLiquidityETH - depends on UniswapV2 router to transfer only the desired amount. Resetting approvals is prohibited due to integration with HBTC where 0 approvals are disallowed.

	removeLiquidity, removeLiquidityETH - could skip reducing allowance if it is set to MAX_UINT.
HWeth	None
HYVault	None

24. Upon calling a DSProxy, the target contract must be an allowed one 🔎

For HMaker, only `getProxyActions()` may be a target of the DSProxy call. For HSCompound, only `FCOMPOUND_ACTIONS()` may be a target of the DSProxy call.

Handler	Violating functions
HMaker	None
HAaveProtocol	N/A
HAaveProtocolV2	N/A
HSCompound	None
HFunds	N/A
HOneInchExchange	N/A
HBProtocol	None
HBalancer	None (applicable to join* functions)
HCEther	N/A
HCToken	N/A
HComptroller	N/A
HCurve	N/A
HCurveDao	N/A
HFurucomboStaking	N/A
HGasTokens	N/A
HKyberNetwork	N/A
HMooniswap	N/A
HStakingRewardsAdapter	N/A
HSushiSwap	N/A
HUniswap	N/A
HUniswapV2	N/A
HWeth	N/A
HYVault	N/A

25. Tokens are moving correctly 🔎

A handler call may *consume* a certain predetermined *amount* of tokens, and may *generate* an unknown number of tokens. Make sure that overall handler execution is indeed satisfying these expectations. *In particular, if there's a token generated, and a positive amount was consumed, we expect a positive amount to be generated.* The modeling of this rule is based on encoding the expected behavior as part of the harness, recording the expected changes according to the document provided by Furucombo.

Handler	Violating functions
HMaker	N/A - DSProxy based
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	N/A - DSProxy based
HFunds	N/A - no intermediate
HOneInchExchange	swap
HBProtocol	N/A - DSProxy based
HBalancer	N/A for join* - DSProxy based, exitswapPoolAmountIn
HCEther	None
HCToken	None
HComptroller	None
HCurve	exchange, exchangeUnderlying
HCurveDao	None
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	swapTokenToToken
HMooniswap	deposit, withdraw (minAmounts could be 0)
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	tokenToTokenSwap
HUniswapV2	None
HWeth	None
HYVault	deposit, withdraw (could compute 0)

26. Balances stay constant for all parties except for the Proxy and the DeFi contract 🔎

Any handler invocation is expected to move tokens between the Proxy contract and an externally owned DeFi protocol (a token/a managing contract). Other balances should stay the same.

Handler	Violating functions
HMaker	N/A - DSProxy based
HAaveProtocol	flashloan (potentially - due to a callback to Proxy)
HAaveProtocolV2	flashloan (potentially - due to a callback to Proxy)
HSCompound	N/A - DSProxy based
HFunds	Expected: inject, sendToken,sendTokens
HOneInchExchange	None
HBProtocol	N/A - DSProxy based
HBalancer	N/A for join* - DSProxy based
HCEther	None
HCToken	None
HComptroller	Expected: claimComp
HCurve	None
HCurveDao	Expected: mint, mintMany
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None

27. [Handler-applicable-1] Stack length increase is bounded 🔎

Handler	Violating functions
HMaker	None
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	None
HFunds	None
HOneInchExchange	None
HBProtocol	None
HBalancer	None
HCEther	None
HCToken	None
HComptroller	None
HCurve	None
HCurveDao	None
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None

CERTORA

28. [Handler-applicable-2] Sender is always cleaned up 🔎

Handler	Violating functions
HMaker	None
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	None
HFunds	None
HOneInchExchange	None
HBProtocol	None
HBalancer	None
HCEther	None
HCToken	None
HComptroller	None
HCurve	None
HCurveDao	None
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None

29. [Handler-applicable-3] Cube counter is always cleaned up 🔎

Handler	Violating functions
HMaker	None
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	None
HFunds	None
HOneInchExchange	None
HBProtocol	None
HBalancer	None
HCEther	None
HCToken	None
HComptroller	None
HCurve	None
HCurveDao	None
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None

CERTORA

30. [Handler-applicable-4] Cache is always cleaned up 🔎

Handler	Violating functions
HMaker	None
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	None
HFunds	None
HOneInchExchange	None
HBProtocol	None
HBalancer	None
HCEther	None
HCToken	None
HComptroller	None
HCurve	None
HCurveDao	None
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None

31. [Handler-applicable-5] “Flat” slots are always cleaned up 🔎

Handler	Violating functions
HMaker	None
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	None
HFunds	None
HOneInchExchange	None
HBProtocol	None
HBalancer	None
HCEther	None
HCToken	None
HComptroller	None
HCurve	None
HCurveDao	None
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None

CERTORA

32. [Handler-applicable-6] “Flat” slots that are written to are not overwritten 🔎

Handler	Violating functions
HMaker	None
HAaveProtocol	None
HAaveProtocolV2	None
HSCompound	None
HFunds	None
HOneInchExchange	None
HBProtocol	None
HBalancer	None
HCEther	None
HCToken	None
HComptroller	None
HCurve	None
HCurveDao	None
HFurucomboStaking	None
HGasTokens	None
HKyberNetwork	None
HMooniswap	None
HStakingRewardsAdapter	None
HSushiSwap	None
HUniswap	None
HUniswapV2	None
HWeth	None
HYVault	None