Secureum CARE Report: **Tracer**

December 2021

CARE: Why

CARE stands for "Comprehensive Audit Readiness Evaluation." CARE is *not* a replacement for a security audit, but is intended to happen before an audit so that protocol code becomes ready for future audit(s) to get a better security outcome from the process.

CARE reviews protocol code mainly for common security pitfalls and best-practices as related to smart contracts written in Solidity specifically for Ethereum blockchain or associated Layer-2 protocols. The pitfalls & best-practices are evaluated from (but not limited to) Secureum's Security Pitfalls & Best Practices 101 and 201.

CARE aims to help identify such common pitfalls & best-practices so that they can be fixed before audit(s). This improves protocol's risk posture earlier in the design & development lifecycle and enables future audit(s) to focus more on deeper/harder application-specific and economic vulnerabilities. CARE helps smart contract security "shift-left" which is widely regarded as significantly improving security posture and outcome.

CARE reviews are performed by "CAREtakers" which includes a Secureum representative (who has a proven track-record of smart contract security expertise/experience) along with invited participants who are top-performing members of the Secureum community and aspiring smart contract security experts. They are invited based on their performance in Secureum's RACEs (i.e. security challenges/quizzes).

CARE: When, Who & What

Dates: December 6th - December 17th, 2021

CAREtakers (Discord handles): Rhynorater#6461, jonah#9279, asgeir.eth#2021, robertistok.eth#3743, luked#8675, Hatchet#3793, StErMi.eth#0277, Kalzak#3920, saian#0610, tqts#0820, RustyRabbit#9674, JM#4592, Hsiu-Ping(Nicholas) Lin#1066, Tadashi#0789, Mai-Hsuan(Kevin) Chia#1071, ospwner#2746, charlesjhongc#4725, PrivacyGeorge#9699, tomasfrancisco#8147, teddav#2297, PlayWith.eth#9252, sarang#5598, Maddiaa#1372, 0xloic#6774, broccolirob#8828, special_m1#8669, lolol#7917 & Rajeev | Secureum#761

Scope: Review focused on this private repository/commit of Tracer protocol: https://github.com/mycelium-ethereum/perpetual-pools-contracts-v2-care/tree/6668af7fd713926065c08ea3a4b3609be3a151f1

Note: Commit hash used is as of December 6th. If there were changes/commits after that across repositories, some of them may have been used in Github Permalinks below but not necessarily reviewed.

CARE Security Pitfalls & Best-Practices: Checklist

1. **Single-step ownership change is risky**
    a. **Summary:** Single-step ownership change for contracts is risky.
    b. **Details:** When privileged roles are being changed, it is recommended to follow a two-step approach: 1) The current privileged role proposes a new address for the change 2) The newly proposed address then claims the privileged role in a separate transaction. This two-step change allows accidental proposals to be corrected instead of leaving the system operationally with no/malicious privileged role. For e.g., in a single-step change, if the current admin accidentally changes the new admin to a zero-address or an incorrect address (where the private keys are not available), the system is left without an operational admin and will have to be redeployed.
    `PoolFactory`, `PoolKeeper` and `PriceObserver` contracts inherit from `Ownable` while `PoolToken` inherits from `ERC20_Cloneable` which implements `transferOwnership` similar to `Ownable`. If the new owner is set to a wrong address this could lead to loss of funds.
    c. **Github Permalinks**: `PoolFactory`, `PoolKeeper`, `PoolToken`, `PriceObserver`
    d. **Mitigation**: Consider replacing single-step ownership change by a two-step change which involves separate propose/grant+claim steps. Also consider overriding the `renounceOwnership()` function to prevent it from ever being called.
2. **Time-delayed change of critical parameters is absent**
    a. **Summary**: Change of critical parameters should be emitted via events and enforced only after a time delay.
    b. **Details**: When critical parameters of systems need to be changed, it is required to broadcast the change via event emission and recommended to enforce the changes after a time-delay. This is to allow system users to be aware of such critical changes and give them an opportunity to exit or adjust their engagement with the system accordingly. For e.g. reducing the rewards or increasing the fees in a system might not be acceptable to some users who may wish to withdraw their funds and exit. This allows a malicious or compromised owner account to change all these critical parameters without giving users time to react.
    `PoolFactory`, `PoolKeeper` and `PriceObserver` contracts inherit from `Ownable` while `PoolToken` inherits from `ERC20_Cloneable` which implements `transferOwnership` similar to `Ownable`. All `onlyOwner` functions that set/change critical addresses/parameters in these contracts should apply a timelock.

c. **Github Permalinks**: `setPoolKeeper`, `setAutoClaim`, `setMaxLeverage`, `setFeeReceiver`, `setSecondaryFeeSplitPercent`, `setFee`, `setMintAndBurnFee`, `setPriceObserver`, `setWriter`

d. **Mitigation**: Critical address/parameter updates should happen only after an adequate time delay so users have time to react. If not, this should be clearly documented and communicated to all users as a potential risk.

3. **Unnecessary constructor**
   a. **Summary:** Proxied contracts with initialisers do not need constructors.
   b. **Details:** Implementation contracts that are deployed by a proxy and initialised via initialisers do not need constructors. `PoolCommitter` contracts are deployed by `PoolFactory` as clones (minimal proxies) with an `initialize` function. But they also have a constructor which is unnecessary.
   Such unnecessary constructors are also present in `PoolToken` and `ERC20_Cloneable`.
   c. **Github Permalinks:** Cloning & Initialising, `PoolCommitter constructor`, `PoolCommitter initialize`, `PoolToken`, `ERC20_Cloneable`
   d. **Mitigation:** Remove the constructor.
4. **Proxy templates not initialised**
   a. **Summary:** Proxied contract templates should also be initialised.
   b. **Details:** `PoolFactory` deployment creates template of `PoolCommitter` contract to be cloned later. The clones will be initialised upon cloning but the original template's `initialize` function is never called and thus left in a state where it can still be called by anyone.
   c. **Github Permalinks:** `PoolFactory`
   d. **Mitigation:** In the `PoolFactory` constructor, initialise the template of `poolCommitterBase`.
5. **Loop may run out of gas**
   a. **Summary**: `PoolCommiter executeCommitments` function has a `while (true)` loop that could run out of gas during execution.
   b. **Details**: `PoolCommiter executeCommitments` function has a `while (true)` loop that calls an gas-intensive function `executeGivenCommitments`. There is a risk that this function fails with an out of gas error if it gets too far behind. The developers note in a comment that "*In reality, this should never iterate more than once,*" but it is not clear that this property has been verified to hold true under different protocol/network conditions.
   c. **Github Permalinks**: `PoolCommitter`
   d. **Mitigation**: Consider modifying `executeCommitments` to a fixed number of iterations.

6. **Name shadowing is misleading**
    a. **Summary**: Name shadowing where two or more variables/functions share the same name could be confusing to developers and/or reviewers.
    b. **Details**:
        i. Function `keeperTip` uses a variable named `keeperTip`. The variable is of type `uint256`, which is the same as the return type of the function.
        ii. Function `deployPairToken` takes the parameter `owner`. This variable shadows an `owner` variable that is inherited from the `Ownable` contract.
    c. **Github Permalinks**: keeperTip, owner
    d. **Mitigation**: Rename the variables to avoid name shadowing.
7. **Return value of functions ignored**
    a. **Summary**: The return value of functions need to be checked for appropriate values/conditions to determine error conditions appropriately. If they are deemed unnecessary, such functions should be declared to not return those values.
    b. **Details:**
        i. The `IOracleWrapper` interface defines a function called `poll` which has a return value of `int256`. `poll` is called once in `PoolKeeper` but its return value is not assigned/used. It appears that `poll` is called solely to modify state.This interface has two implementations, both of which return these values.
        ii. The `IPriceObserver` interface defines a function called `add` which has a return value of `bool`. `add` is called once in `SMAOracle` but its return value is not assigned/used. It appears that `add` is called solely to modify state.This interface has an implementation that returns a boolean value.
    c. **Github Permalinks:** PoolKeeper, SMAOracle
    d. **Mitigation:** Change the interface and implementations to return nothing.
8. **Missing input-validation may lead to infinite loop**
    a. **Summary**: `LeveragedPool.initialize` does not check that `updateInterval` is not zero.
    b. **Details**: `LeveragedPool.initialize` does not check that `updateInterval` is not zero which could cause the loop in `PoolCommitter.executeCommitments` to run unbounded and out of gas if this is accidentally set to 0.
    c. **Github Permalinks**: updateInterval, executeCommitments
    d. **Mitigation**: Add a `require` in `LeveragedPool.initialize` to check that `initialization._updateInterval != 0`.
9. **Missing sanity/threshold checks in input validation**

a. **Summary**: Sanity/threshold checks (depending on their types) on input parameters are missing.
b. **Details**:
    i. `poolCommitterAddress` is not validated to be a valid pool committer address in `AutoClaim.paidClaim`.
    ii. `frontRunningInterval` and `updateInterval` may be too short (even 0) or too long. If the variable is incorrectly set during a new pool deployment, it can leave the pool unprotected against front running attacks or the pool may be unusable due to an unreasonably long interval.
    iii. No sanity lower/upper bound check on `_price` in `setGasPrice`.
    iv. No sanity check on `_updateIntervals` in `SMAOracle` constructor.
    v. No sanity check that new `_governance != governance` (current).
c. **Github Permalinks**: `AutoClaim.paidClaim`, `frontRunningInterval`, `setGasPrice`, `SMAOracle`, `transferGovernance`
d. **Mitigation**: Perform appropriate sanity/threshold checks (depending on their types) on input parameters.

10. Incorrect order of input validation

a. **Summary:** Input validation on parameters is performed *after* their values are used in function.
b. **Details:** The input validation of `deployParameters`, `leverageAmount` and `quoteToken` is done after the values are used in the creating clone contracts. Similarly, the threshold validation of `secondaryFeeSplitPercent` is done in `feeTransfer` instead of `initialize`. Gas used will unnecessarily be more when input validation fails.
c. **Github Permalinks:** `deployPool`, `feeTransfer`
d. **Mitigation:** Perform input validation before values are used in function.

11. Unnecessary `initialize` can help bypass access control

a. **Summary**: Unnecessary `initialize` function in `AutoClaim` can help bypass access control.
b. **Details**: There will be (at most) one instance of `AutoClaim` which means that this is not meant to be behind a proxy which requires (atomic) initialization via deployment script or factory. However, there is an `initialize` function implemented which has `external` visibility and that sets the `poolFactory` address.
Given that this `initialize` function is never called by the protocol's typical flows, it may be called once (because of the `initializer` modifier) by an attacker to set the `poolFactory` address to their controlled malicious contract. If that malicious contract implements a dummy version of `poolFactory.isValidPoolCommitter` used in the `onlyPoolCommitter` modifier, it can be used to bypass access control on functions `makePaidClaimRequest` and `withdrawUserClaimRequest`.

c. **Github Permalinks**: `initialize`, `onlyPoolCommitter`, `makePaidClaimRequest`, `withdrawUserClaimRequest`

d. **Mitigation**: Remove `initialize` function.

12. **Gas optimizations**

a. **Summary:** There are various best-practices to make code more gas efficient.

b. **Details:**

i. Variables set only once in constructor/initialize can be made immutable.

ii. `PriceObserver.leftRotateWithPad` O(n) algorithm could be optimised to O(1). Instead of moving every element in the array, simply maintain a pointer to which index is the oldest in the array. When a new item is added, replace the oldest index and update: pointer = (pointer + 1) % length. Note that this does not affect the SMA algorithm which uses `getAll`. Although the elements will be in a different order, addition is commutative. Additionally, the SMA algorithm can also be reduced to O(1) by having the `PriceObserver` keep a running total of items (subtract items that get evicted, add new ones) and return that sum instead of O(n) summing all of the items every time an average is calculated.

iii. Inherit from `InvariantCheck` in `LeveragedPool` and `PoolCommitter` instead of making external calls.

iv. Pass `owner` from `PoolFactory` to the `PoolCommitter` instead of having it call back in `initialize`.

v. `poolBase` and `poolCommitterBase` are unnecessary in the presence of `poolBaseAddress` and `poolCommitterBaseAddress`.

vi. Consider optimising `multiPaidClaimMultiplePoolCommitters` by batching all the payments in a single `transfer`.

vii. Replace two external calls to `ILeveragedPool(_poolAddress).oracleWrapper` and `IOracleWrapper(oracleWrapper).getPrice` with a single one `ILeveragedPool(_poolAddress).getOraclePrice` in `PoolKeeper.newPool`.

viii. State variables read multiple times can be cached in local variables.

ix. Variables whose initial values are expected to be the same as default values of their types can skip explicit initialisation to save gas.

x. Expressions where the written value is guaranteed to not overflow/underflow can use `unchecked{}` blocks to skip inbuilt compiler bounds check to save gas.

c. **Github Permalinks:**

i. `updateInterval`, `oracle`, `scaler`

ii. `PriceObserver.leftRotateWithPad`

iii. `InvariantCheck`, `LeveragedPool`, `PoolCommitter`

iv. `initialize`

v. `poolBaseAddress` and `poolCommitterBaseAddress`
vi. `multiPaidClaimMultiplePoolCommitters`
vii. `PoolKeeper.newPool`
viii. `provisionalGovernance` and `governance`, `autoClaim`, `updateIntervalId`, `paused`, `quoteToken`, `feeAddress`, `keeper`
ix. `lastUpdate`, `updateInterval`, `S`, `observer`, loop index `i`, `numElems`, `writer`
x. `updateIntervalId`, `counter`, `numPools`, `remainder`

d. **Mitigation:** As noted in the respective details.

13. Missing explicit visibility for state variables

a. **Summary:** Visibility of important protocol parameters can be made public.
b. **Details:** Certain protocol parameters captured in state variables have the default `internal` visibility without any getter functions. It will be helpful to either change their visibility to public or add public getters to allow users to read their values easily.
   i. `lastUpdate` and `updateInterval` in `SMAOracle`
   ii. `mintingFee` and `burningFee` in `PoolCommiter`
c. **Github Permalinks:** `mintingFee` and `burningFee`, `lastUpdate` and `updateInterval`
d. **Mitigation:** Evaluate visibility of protocol parameters.

14. Modifier with side-effects

a. **Summary:** Modifiers with side-effects are risky.
b. **Details:** Modifiers should only implement checks and not make state changes and external calls which violates the checks-effects-interactions pattern. These side-effects may go unnoticed by developers/reviewers because the modifier code is typically far from the function implementation.
   Modifier `updateBalance` calls `updateAggregateBalance` before the modifier's function executes, which adds the user's most recent commit to their aggregated balance. This modifier is used only once by the `PoolCommiter` `commit` function.
c. **Github Permalinks:** `updateBalance`, `updateAggregateBalance`, `commit`
d. **Mitigation:** Replace modifier by an internal function which gets called at the beginning of the function.

15. Public function visibility can be made external

a. **Summary**: Functions should have the strictest visibility possible. Public functions may lead to more gas usage by forcing the copy of their parameters to memory from calldata.

b. **Details**: If a `public` function is never called from the contract it should be marked as `external`. This will save gas.
c. **Github Permalinks**: `withdrawUserClaimRequest`, `checkUserClaim`, `getAggregateBalance`, `keeperTip`, `get`, `getAll`, `add`, `setWriter`, `getWriter`, `clear`, `getLastUpdate`
d. **Mitigation**: Change visibility from `public` to `external`.

16. **Avoid `transfer` as reentrancy mitigation**
    a. **Summary:** `transfer` as reentrancy mitigation may break with gas repricing of opcodes.
    b. **Details:** Although `transfer` has been recommended in the past as a security best-practice to prevent reentrancy attacks because it only forwards 2300 gas, gas repricing of opcodes may break deployed contracts. There are four places where `transfer` is used where checks-effects-interactions pattern is not followed.
    c. **Github Permalinks:** `makePaidClaimRequest`, `paidClaim`, `withdrawClaimRequest`, `withdrawUserClaimRequest`
    d. **Mitigation:** Consider using OpenZeppelin's `sendValue` or use `call` without hardcoded gas limits along with checks-effects-interactions pattern and/or reentrancy guards for reentrancy protection.
17. **Costly operations inside a loop**
    a. **Summary:** Operations such as state variable updates and external calls cost a lot of gas, are expensive and may lead to out-of-gas errors when performed inside a loop especially if the loop index can be user-controlled.
    b. **Details:** Programming patterns such as looping over arrays of unknown size may lead to out-of-gas errors when the gas cost of execution exceeds the provided transaction gas or the block gas limit.
    c. **Github Permalinks:** `multiPaidClaimMultiplePoolCommitters`, `multiPaidClaimSinglePoolCommitter`
    d. **Mitigation:** Avoid expensive operations within loops and check that the loop index cannot be user-controlled or is bounded.
18. **Modifiers `checkInvariantsBeforeFunction` and `checkInvariantsAfterFunction` undermine contract pausing**
    a. **Summary:** Modifiers `checkInvariantsBeforeFunction` and `checkInvariantsAfterFunction` check for invariants and set a boolean `paused` if any invariant does not hold. However, the same modifiers trigger a revert if boolean `paused` is set after checking for invariants.
    b. **Details:** Modifiers `checkInvariantsBeforeFunction` and `checkInvariantsAfterFunction` in `LeveragedPool` and `PoolCommitter` check for invariants in

`InvariantCheck.checkInvariants`, set a boolean `paused` and emit a `Paused` event if any invariant does not hold. However, the same modifiers trigger a revert if boolean `paused` is set after checking for invariants. This effectively reverts any transaction that violates invariants instead of enabling pausing of contracts.

c. **Github Permalinks:** `LeveragedPool modifiers`, `PoolCommitter modifiers`, `InvariantCheck.checkInvariants`

d. **Mitigation:** Specify the intended pause/unpause behaviour on invariant failure and implement accordingly. Also, modifiers should not change the state of contracts.

19. Ether can get locked in contract

a. **Summary:** Ether can get locked in the `PoolCommitter` contract when a user mistakenly sets the `payForClaim` parameter to `false` while sending Ether to the `commit` function.

b. **Details:** The `PoolCommitter commit` function is declared `payable`. However, a user failing to set `payForClaim` to `true` while sending Ether to the `commit` function will bypass the call to forward `msg.value` to `autoClaim.makePaidClaimRequest`. That Ether deposit is locked in the `PoolCommitter` contract with no way to get it out.

c. **Github Permalinks:** `PoolCommitter.commit`

d. **Mitigation:** Add a check that enforces `msg.value > 0 && payForClaim` or an access controlled function to sweep any such locked Ether from the `PoolCommiter` contract.

20. Missing events for critical functions

a. **Summary**: Critical functions do not emit events.

b. **Details**: Events should be emitted for critical functions that are access-controlled, setters of protocol parameters or those affecting protocol in significant ways:

i. `ERC20_Cloneable.transferOwnership`.

ii. `Paused` and `Unpaused` in `PoolCommiter.pause` and `PoolCommiter.unpause`.

iii. `RequestWithdrawn` in `withdrawUserClaimRequest`.

iv. `withdrawQuote` which is used by `onlyGov` to withdraw all quote assets when paused.

v. Privileged setters: `setQuoteAndPool`, `setWriter`, `setFactory`, `setPriceObserver`, `setGasPrice`, `setNewPoolBalances`, `payKeeperFromBalances`, `setPoolKeeper`, `setMaxLeverage`, `setFeeReceiver`, `setSecondaryFeeSplitPercent`, `setFee`, `setMintAndBurnFee`.

vi. When the user self-claims in `makePaidClaimRequest`, emission of `PaidRequestExecution` is missing.

c. **Github Permalinks**: ERC20_Cloneable.transferOwnership, PoolCommiter.pause, PoolCommiter.unpause, withdrawUserClaimRequest, LeveragedPool.withdrawQuote, PoolCommitter.setQuoteAndPool, PriceObserver.setWriter, PoolKeeper.setFactory, PoolKeeper.setPriceObserver PoolKeeper.setGasPrice, LeveragedPool.setNewPoolBalances, LeveragedPool.payKeeperFromBalances, PoolFactory.setPoolKeeper, PoolFactory.setMaxLeverage, PoolFactory.setFeeReceiver, PoolFactory.setSecondaryFeeSplitPercent, PoolFactory.setFee, PoolFactory.setMintAndBurnFee, PaidRequestExecution

d. **Mitigation**: Emit appropriate events with necessary parameters.

21. **Divide before multiply may lead to loss of precision**

a. **Summary**: `_tipPercent` is divided by 100 before being multiplied by `_keeperGas`.

b. **Details**: Performing multiplication before division is generally better to avoid loss of precision because Solidity integer division may truncate.

c. **Github Permalinks**: PoolKeeper

d. **Mitigation**: Consider performing multiplication before division.

22. **`block.timestamp` used as time proxy**

a. **Summary**: Risk of using `block.timestamp` for time should be considered.

b. **Details**: `block.timestamp` is not an ideal proxy for time because of issues with synchronisation, miner manipulation and changing block times.

c. **Github Permalinks**: LeveragedPool, PoolCommitter, PoolKeeper, SMAOracle

d. **Mitigation**: Consider the risk of using `block.timestamp` as time proxy and evaluate if block numbers can be used as an approximation for the application logic. Both have risks that need to be factored in.

23. **Token approval may be unnecessary and fail**

a. **Summary**: Token approval may be unnecessary and also not compatible with some ERC20 tokens.

b. **Details**: `setQuoteAndPool` grants `LeveragedPool` unlimited approval to the quote tokens it owns but it is not clear that it needs to own any quote tokens. Also, some ERC20 tokens may require allowance to be zero before calling `approve` with a non-zero value.

c. **Github Permalinks**: PoolCommitter

d. **Mitigation**: Evaluate the need for this approval. Consider setting allowance to 0 before approving to a different value or use `safeIncreaseAllowance` which is not susceptible to the `approve` race condition.

24. **Block time may not be a constant**

a. **Summary**: Block time may not be a constant value of 13 as used.

b. **Details**: The keeper tip is increased by `TIP_DELTA_PER_BLOCK` every block. For this, the block time is assumed to be a constant value of 13 seconds. While this value fluctuates every block, it is expected to become 12 seconds after the anticipated Merge of the Ethereum protocol. Under Proof-of-Stake which comes into effect after the Merge, blocks will be created exactly 12 seconds apart.
Keeper's tip is calculated as `BASE_TIP + (TIP_PER_BLOCK * elapsedBlocks) / BLOCK_TIME`. If average block time increases, this would result in keepers getting overpaid because they now do upkeep less frequently than before. If average block time decreases (e.g. Merge), this would result in reduced keeper incentive which may make them do upkeep less frequently.

c. **Github Permalinks**: `BLOCK_TIME`, `keeperTip`

d. **Mitigation**: Consider using `block.number` or factor in the anticipated change in block times in other ways e.g. change BLOCK_TIME variable from constant to a state variable with a setter to change it after Merge.

25. Use of magic numbers is confusing and risky

a. **Summary**: Magic numbers are hardcoded numbers used in the code which are ambiguous to their intended purpose. These should be replaced with constants to make code more readable and maintainable.

b. **Details**:
   i. 365: Annual fee is divided by 365 days to calculate fee per update. Leap years have 366 days. 365 is not guaranteed to be the number of days every year as assumed. Depending on the precision required, 365.25 or 365.2425 days per year is more accurate.
   ii. Long and Short tokens are accessed in `PoolCommitter` code using indices 0 and 1 instead of using the constants `LONG_INDEX` and `SHORT_INDEX` defined for them.

c. **Github Permalinks**:
   i. `365 days`, `PoolCommiter`

d. **Mitigation**: Replace magic hardcoded numbers with declared constants.

26. Constant not used consistently everywhere

a. **Summary:** Constant `MAX_NUM_ELEMS` set to 24 is not used consistently everywhere.

b. **Details:** The last MAX_NUM_ELEMS oracle prices are stored in an array but this constant is not used in some places. For example, functions `SMA` and `getAll` use a hard-coded value of 24 for the size of their array parameters.

c. **Github Permalinks:** `MAX_NUM_ELEMS`, `SMA`, `getAll`

d. **Mitigation:** Use the constant `MAX_NUM_ELEMS` consistently in all places instead of the hard-coded value 24.

27. Unnecessary inheritance

a. **Summary:** `Context` is inherited twice.

b. **Details:** `Context` is inherited from ERC20 as well as directly in `ERC20_Cloneable`. This is unnecessary and can be removed.

c. **Github Permalinks:** `ERC20_Cloneable`

d. **Mitigation:** Remove the direct inheritance of `Context`.

28. Code, comments and documentation

a. **Summary**: Discrepancies in/between or inaccuracies/deficiencies in code, comment and documentation can be misleading and could indicate the presence of inaccurate implementation or documentation.

b. **Details:**

i. Specification and documentation have not been updated for V2 of the protocol which is significantly complex. No descriptive changelog from V1 to V2 is available.

ii. Natspec of `PoolToken.mint` `@param` refers to `burn`.

iii. Missing Natspec for `ChainlinkOracleWrapper.poll`.

iv. Document that only ERC20 tokens that implement the optional `decimals` function will be accepted in the protocol.

v. Natspec comment for `LeveragedPool.poolTokenTransfer` says long tokens but function handles both short and long tokens depending on boolean parameter `isLongToken`.

vi. Natspec comments for `LeveragedPool.mintTokens` and `LeveragedPool.burnTokens` say `isLongToken` should be `true` if minting/burning *short* token but it should really be *long* token.

vii. Natspec for `applyCommitment` incorrectly mentions `@param userCommit` twice. The second one should really be `@param totalCommit`.

viii. Duplicated comments in `deployPool`.

ix. Incomplete `@notice` comment in `withdrawUserClaimRequest`.

x. Address or remove `@dev` comment in the `AutoClaim` contract comment block.

xi. Missing comment above the `SMAOracle` contract.

xii. Unused comment title “Global state” in `PoolToken`.

xiii. The param `poolCommitterAddress` is of address type while `@dev` comment treats it as a list of addresses.

xiv. The param `owner` is not documented in `deployPairToken`.

xv. The param `payForClaim` is not documented in `commit`.

xvi. Tokens mentioned in `@amount` and `@return` should be pool tokens and settlement tokens respectively, not the other way around. The function should be `getBurn` not `getMint`.

xvii. Comment of `PoolToken.mint` should say mint, not burn.

xviii. `@dev` note of `withdrawQuote` states “Pool must not be paused” while in the function it has `require(paused)`.

xix. Code and comment mismatch when `k` is equal to `xs` length because `SMA @dev` says “*Throws if `k` is greater than or equal to the length of `xs` (due to buffer overrun potential)*” while the code implements `require(... && k <= n &&...)`.

xx. Code and comment mismatch when `k` is equal to `uint256(type(int256).max)` because `SMA @dev` says “*Throws if `k` is the maximum *signed* 256-bit integer (due to necessary division)*” while the code implements `k <= uint256(type(int256).max)`.

xxi. Comment for modifier in `LeveragedPool.checkInvariantsBeforeFunction` refers incorrectly to `PoolCommitter`.

xxii. Documentation says that pool deployer gets to specify the pool owner but the implementation sets this to the `PoolFactory` owner which is the governance.

xxiii. Incorrect comment specifying an initialiser address that does not exist in: “*@dev Only callable by the associated initialiser address*” in `PoolCommitter.initialize`.

xxiv. Missing Natspec @param’s for `ChainlinkOracleWrapper.constructor`, `LeveragedPool.initialize`, `PoolFactory.constructor`, `PoolKeeper.constructor`, `SMAOracle.constructor`, `PoolSwapLibrary.mulFraction`.

xxv. `ERC20_Cloneable` constructor `@dev` comments incorrectly specify a deprecated `_setupDecimals` function and that `decimals` has 18 as default value.

xxvi. Comment for `address _invariantCheckContract` in `ILeveragedPool` references `PoolCommitted` while address is for `invariantCheckContract`.

xxvii. Comment for `poolName` says “pool code”.

xxviii. Missing Natspec for `IPoolCommitter` functions and structs.

xxix. Missing Natspec for `IPoolFactory` and `IPoolKeeper`.

xxx. Out of order `@params` compared to parameters in function declaration in `LeveragedPool.poolTokenTransfer`.

xxxi. The comment: "*// Since lastPriceTimestamp <= block.timestamp, the below also confirms that timestamp >= block.timestamp*" is incorrect because `lastPriceTimestamp <= block.timestamp` and `lastPriceTimestamp <= timestamp` does not always imply that `block.timestamp <= timestamp` because this can also be satisfied when `lastPriceTimestamp <= timestamp <= block.timestamp`.

xxxii. Comment says: "*Assumes shortBalance + longBalance >= reward*" but code implements reward `shortBalance + _longBalance > amount` while it should be `'>='`.

xxxiii. Comment says: "*@return Quantities of pool tokens for this pool (long and short, respectively)*" but it should be "*short and long*".

xxxiv. Comment "*// Modifiers*" should be above `modifier checkInvariantsBeforeFunction`.

xxxv. Comment in `keeperGas` says: "*safe due to explicit bounds check above*" before making an explicit downcast from `uint256` to `bytes16` but it is not clear which bounds check is being referred to.

c. **Github Permalinks:** `PoolToken.mint`, `ChainlinkOracleWrapper.poll`, `LeveragedPool.poolTokenTransfer`, `LeveragedPool.mintTokens`, `LeveragedPool.burnTokens`, `PoolCommitter.applyCommitment`, `PoolFactory.deployPool`, `withdrawUserClaimRequest`, `AutoClaim`, `SMAOracle`, `PoolToken`, `poolCommitterAddress`, `deployPairToken`, `commit`, `PoolSwapLibrary.getBurn`, `PoolToken.mint`, `withdrawQuote`, `SMA k and xs length`, `SMA and max integer`, `LeveragedPool.checkInvariantsBeforeFunction`, `PoolFactory.owner`, `PoolCommitter.initialize`, `ChainlinkOracleWrapper.constructor`, `LeveragedPool.initialize`, `PoolFactory.constructor`, `PoolKeeper.constructor`, `SMAOracle.constructor`, `PoolSwapLibrary.mulFraction`, `ERC20_Cloneable.constructor`, `ILeveragedPool._invariantCheckContract`, `ILeveragedPool.poolName`, `IPoolCommitter functions`, `IPoolCommitter structs`, `IPoolFactory`, `IPoolKeeper`, `LeveragedPool.poolTokenTransfer`, `lastPriceTimestamp <= timestamp <= block.timestamp`, `shortBalance, longBalance, reward` Comment, `shortBalance, longBalance, reward` Code, short and long, `modifier checkInvariantsBeforeFunction`,

keeperGas

d. **Mitigation:** Code, comments and documentation should all be accurate and consistent.

29. **Ordering of parameters**
    a. **Summary:** `PoolToken` `mint` and `burn` parameters are switched from the typical order.
    b. **Details:** Token `mint` and `burn` functions typically take two parameters: address and amount in that order. However, the `PoolToken` `mint` and `burn` functions have this reversed which could be confusing and lead to errors.
    c. **Github Permalinks:** `PoolToken.mint`
    d. **Mitigation:** Correct the order to what is typically used.
30. **Missing zero-address validation**
    a. **Summary**: Missing zero-address validation for address parameters may lead to ETH/tokens getting burned or cause reverts.
    b. **Details**: The following address parameters are missing zero-address checks:
        i. `secondaryFeeAddress`: Missing zero address check for this in `initialize` seems to be to allow primary fee address DAO to collect all fees. However, given that this can only be updated by the previous `secondaryFeeAddress` and not `onlyGov`, a zero address used for initialization accidentally cannot be changed later, requiring contract redeployment.
        ii. `deployer` in `ChainlinkOracleWrapper` constructor.
        iii. `factory` in `setFactory`.
        iv. `oracle`, `observer` and `deployer` in `SMAOracle` constructor.
    c. **Github Permalinks**: `secondaryFeeAddress`, `ChainlinkOracleWrapper`, `PoolKeeper.setFactory`, `SMAOracle`
    d. **Mitigation**: Consider adding zero-address validation with a require statement or custom errors.
31. **Missing indexed event parameters**
    a. **Summary**: Events without indexed event parameters make it harder and inefficient for off-chain tools to analyse them.
    b. **Details**: Indexed parameters (“topics”) are searchable event parameters. They are stored separately from unindexed event parameters in an efficient manner to allow for faster access. This is useful for efficient off-chain-analysis, but it is also more costly gas-wise.
    `PaidRequestExecution` and `RequestWithdrawn` events do not use indexed parameters.
    c. **Github Permalinks**: `PaidRequestExecution`, `RequestWithdrawn`
    d. **Mitigation**: Consider which event parameters could be particularly useful to off-chain tools and should be indexed.
32. **Unnecessary modifiers**

a. **Summary:** Unnecessary modifier `checkInvariantsAfterFunction` on functions.
b. **Details:** `checkInvariantsAfterFunction` is used to check invariants on pool balance and pending mints. None of the functions: `updateFeeAddress`, `setKeeper`, `transferGovernance` or `claimGovernance` affect any of those invariants but have the modifier applied which leads to unnecessary computation and wasted gas.
c. **Github Permalinks:** `updateFeeAddress`, `setKeeper`, `transferGovernance`, `claimGovernance`
d. **Mitigation:** Consider if these modifiers can be removed safely.

33. Potential lock of funds

a. **Summary**: Funds may be locked in the protocol if contracts that cannot receive Ether interact with it.
b. **Details**: If interacting contracts have no `payable` fallback/receive functions, the transferred Ether will be locked because the transaction will revert.
c. **Github Permalinks**: `makePaidClaimRequest`, `paidClaim`, `withdrawClaimRequest`, `withdrawUserClaimRequest`
d. **Mitigation**: Consider changing to a pull payments model.

34. Unused named returns

a. **Summary:** Mixing of named and explicit returns is confusing especially when named returns are unused.
b. **Details:** `_latestRoundData` and `getPriceAndMetadata` functions have named returns that they do not use in favour of returning different variables explicitly. This decreases code readability and may be confusing for developers/reviewers.
c. **Github Permalinks:** `_latestRoundData`, `getPriceAndMetadata`
d. **Mitigation:** Reconsider the use of named returns and instead explicitly use return statements with values to increase readability.

35. Missing support/documentation for use of deflationary tokens

a. **Summary:** Deflationary tokens can break the accounting model of the system.
b. **Details:** Deflationary tokens deduct a fee in transfers which causes their amount transferred to be less than that specified. It is not clear that the contracts plan to support such tokens (via a whitelist for e.g.) as quote tokens because the implemented logic does not calculate the before-and-after balances to account for such tokens. For e.g., USDT has the ability to collect fees that is not activated now, but may do so in the future. Allowed use of such deflationary tokens in the protocol may lead to under-collateralized pools because of incorrect accounting.
c. **Github Permalinks:** `pool.quoteTokenTransferFrom`
d. **Mitigation:** Do not allow deflationary quote tokens (e.g. maintain an allow list as part of guarded launch). Alternatively, add logic to support such tokens or

document the non-support warning.

36. **Rebasing quote token can result in failed invariant check**
    a. **Summary**: If the quote token of the pool is a token that is routinely rebased, it can result in failed invariant checks and contracts getting paused.
    b. **Details**: In `InvariantCheck.checkInvariants`, it checks that the quote token balance of the pool is greater than the sum of pool tokens and pending mints. If quote token is rebased, it can result in its balance being decreased and hence result in failed invariant check and contracts getting paused. Allowed rebasing tokens in the protocol may also lead to under-collateralized pools because of incorrect accounting.
    c. **Github Permalinks**: InvariantCheck.checkInvariants
    d. **Mitigation**: Do not allow quote tokens that do rebasing (e.g. maintain an allow list as part of guarded launch). Alternatively, add logic to support such tokens or document the non-support warning.
37. **Unused constructs**
    a. **Summary**: Unused constructs (interfaces, variables, functions, modifiers, events etc.) indicate missing or unnecessary functionality.
    b. **Details**:
        i. Interface `IHistoricalOracleWrapper` is not implemented or used by any contract.
        ii. Constant `MAX_DECIMALS` in PoolKeeper is unused.
        iii. Event definitions in `IInvariantCheck`: `InvariantsHold` and `InvariantsFail` are not used for any emit.
        iv. Modifier `onlyGov` in `PoolFactory` is declared but unused with any function.
        v. The `deployer` variable in `ChainlinkOracleWrapper` contract is not used.
    c. **Github Permalinks**: `IHistoricalOracleWrapper`, `MAX_DECIMALS`, `InvariantsHold`, `InvariantsFail`, `PoolFactory.onlyGov`, `ChainlinkOracleWrapper.deployer`
    d. **Mitigation**: Evaluate if unused constructs indicate missing functionality that needs to be implemented or remove them to improve readability and auditability.
38. **Checks-Effects-Interactions (CEI) pattern is not followed**
    a. **Summary**: Not following CEI pattern is risky in the reentrancy context.
    b. **Details**: Following CEI pattern is a best-practice even if external interactions with contracts are trusted. The protocol interacts with arbitrary tokens that may not be within the trust model. Such tokens or pool creators may be malicious, forcing reentancies where sufficient mitigations are not in place.
    c. **Github Permalinks**: `PoolCommitter.commit`, `PoolCommitter.claim`, `AutoClaim.makePaidClaimRequest`, `AutoClaim.paidClaim`, `AutoClaim.withdrawClaimRequest`, `AutoClaim.withdrawUserClaimRequest`, `PoolFactory.deployPool`
    d. **Mitigation**: Follow best-practice to adhere to the CEI pattern wherever possible and use reentrancy guard wherever required.
39. **Incorrect data deleted leading to logic error**

a. **Summary:** Cleaning up incorrectly/insufficiently will lead to reuse of stale state and/or removal of valid state, both of which may lead to security issues.
b. **Details:** `PoolCommitter.updateAggregateBalance` is incorrectly using the loop index `i` instead of the interval `id` at that index in `currentIntervalIds` while deleting `userCommitments` and `unAggregatedCommitments` for an user.
c. **Github Permalinks:** `PoolCommitter.updateAggregateBalance`
d. **Mitigation:** Replace loop index `i` with interval `id`.

40. Return value of functions ignored

a. **Summary**: The return value of functions need to be checked for appropriate values/conditions to determine error conditions appropriately.
b. **Details**: The return values of `priceObserverWriter.poll` and `priceObserver.add(latestPrice)` are ignored to only affect state change.
c. **Github Permalinks**: `priceObserverWriter.poll`, `priceObserver.add`
d. **Mitigation**: Consider checking return values of the functions to detect error codes/conditions and verify expected values.

41. Redundant variable assignment

a. **Summary:** `PoolCommitter governance` address is initialised twice.
b. **Details:** In the `PoolCommitter initialize` function, `governance` address is initialised twice: once with `getOwner` on the function parameter `_factory` and again with `getOwner` on the state variable `factory` which was previously initialised with function parameter `_factory`. The second assignment is redundant.
c. **Github Permalinks:** `governance`
d. **Mitigation:** Remove the second redundant assignment.

42. Use of uninitialised variables

a. **Summary:** Using uninitialised variables may result in undefined behaviour.
b. **Details:** `PoolKeeper.observer` address is not initialised in the constructor. However, it is used in `performUpkeepSinglePool` directly without a zero address check with the assumption that its setter `setPriceObserver` will be called by the owner before that. Without that initialisation, it will lead to reverts.
c. **Github Permalinks:** `observer`, `constructor`, `performUpkeepSinglePool`, `setPriceObserver`
d. **Mitigation:** Initialise `observer` in constructor or add zero address check in `performUpkeepSinglePool`.

43. **Redundant modifiers**
    a. **Summary**: The `poolUpKeep` flow has repeated `checkInvariantsBeforeFunction` modifiers which increases gas cost and decreases code readability and maintainability.
    b. **Details**: `LeveragedPool.executePriceChange` has a `checkInvariantsBeforeFunction` and `PoolCommitter.executeCommitments` also has a `checkInvariantsBeforeFunction`. They are executed in sequence in `LeveragedPool.poolUpkeep`.
    c. **Github Permalinks**: checkInvariantsBeforeFunction in executePriceChange, checkInvariantsBeforeFunction in executeCommitments, poolUpkeep
    d. **Mitigation**: Remove one of the `checkInvariantsBeforeFunction` to reduce gas cost and improve code readability and maintainability.
44. **Missing equality check in setter functions**
    a. **Summary:** Setter functions should check if the new value is different from the old value.
    b. **Details:** `LeveragedPool.updateSecondaryFeeAddress` does not check if the new address is different from the old address. This will allow accidental setting of the same address to potentially cause a mismatch between on-chain state and external bookkeeping, and also cause event emission spam/confusion.
    c. **Github Permalinks:** LeveragedPool.updateSecondaryFeeAddress
    d. **Mitigation:** For all setters, check if the new value is different from the old value and revert otherwise.
45. **Malicious/Flawed `InvariantCheck` contract can lock funds**
    a. **Summary:** A malicious pool creator can lock user funds via a malicious `InvariantCheck` contract.
    b. **Details:** Tracer protocol allows anyone to deploy a pool and specify their own `invariantCheckContract` contract (as part of the deployment parameters) which is used to check for the violation of invariants within the pool. This contract is called at the end of almost every function via `checkInvariantsBeforeFunction` and `checkInvariantsAfterFunction` modifiers and is expected to pause `LeveragedPool` and `PoolCommitter` contracts on invariant failure (although this incorrectly reverts for now). This gives pool creator the ability to pause the pool (requiring an unpause by governance) at will depending on any triggers in the pool or `msg.sender`. Note that this may also happen accidentally due to a benign user accidentally using a flawed `InvariantCheck` contract.
    Pausing the pool arbitrarily will not allow users to adjust their position as

intended, which may result in loss for users either via overstaying their position or via volatility delay.

c. **Github Permalinks:** `PoolFactory.deployPool`

d. **Mitigation:** Do not allow users to specify their own invariant contracts. Alternatively, factor in the risk from such malicious/misbehaving contracts.

46. **`getOwner` duplicates existing functionality**

a. **Summary**: `PoolFactory` `getOwner` function duplicates OpenZeppelin's `owner` function.

b. **Details**: The `getOwner` function is explicitly defined in the `PoolFactory` contract. OpenZeppelin's `Ownable` which is inherited in `PoolFactory` already contains this getter in the form of `owner`.

c. **Github Permalinks**: `getOwner`

d. **Mitigation**: Remove `getOwner` function and replace its usage by `owner`.

47. **Contract addresses as parameters**

a. **Summary:** Contract addresses are not checked for existence or against expected interfaces.

b. **Details:** Contract addresses passed as parameters during construction/initialisation are checked for zero addresses. However they are not checked for existence or to support the required interface.

c. **Github Permalinks**: `LeveragedPool`

d. **Mitigation:** Consider checking the existence of contracts and using `ERC165` style `supportsInterface` functions in those contracts to check whether the passed addresses implement those interfaces.

48. **A claim can be invoked needlessly**

a. **Summary:** A claim can be invoked without successfully claiming or getting rewards but wasting gas in the process.

b. **Detail:** There is no easy way for users and auto-claimers to check if a claim indeed can be claimed and if there is a reward associated with it. Without that, a needlessly attempted claim can return without claiming or getting associated rewards but only wasting the caller's gas.

c. **Github Permalinks**: `PoolCommitter.claim`

d. **Mitigation**: Implement an external view method to allow users to check for claims and associated rewards.

49. **Pay for claim without rewards**

a. **Summary:** Users can call `commit` with `payForClaim` enabled but without sending any Ether.

b. **Details:** Users can call `commit` with `payForClaim` enabled but without sending any Ether i.e. no reward. This is possible because there is no check for `msg.value > 0` when `payForClaim` is true.

c. **Github Permalinks:** `commit`

d. **Mitigation:** Add check for `msg.value > 0` when `payForClaim` is true.

50. `newPool` could revert

a. **Summary:** `newPool` could revert if pool oracle is a `SMAOracle` without enough historical data to calculate SMA.

b. **Details:** When a new pool is deployed by the `PoolFactory`, if the oracle used by the pool is a `SMAOracle` that has not yet accumulated the required historical data to cover the SMA periods, it will revert on `getPrice` and revert pool deployment.

c. **Github Permalinks:** `newPool`

d. **Mitigation:** During deployment, check if oracle is a `SMAOracle` that has accumulated the required historical data to cover SMA periods.

51. Pools and `PoolKeeper` may use different oracles

a. **Summary:** There is no enforcement that pools and `PoolKeeper` use the same oracles and same type of oracles: `SMAOracle` vs `SpotOracle`.

b. **Details:** Pools and `PoolKeeper` using different oracles or different types of oracles (`SMAOracle` vs `SpotOracle`) may lead to mismatched assumptions on prices and updates leading to incorrect bookkeeping and effects.

c. **Github Permalinks:** `setPriceObserver`, `LeveragePool Init`

d. **Mitigation:** Enforce the use of the same oracle by not passing the `oracleWrapper` as a parameter for `LeveragedPool` but directly using the oracle used by the keeper.

52. `performUpkeepSinglePool` may revert

a. **Summary:** `performUpkeepSinglePool` may revert on the second call if it is not called after at least `SMAOracle.updateInterval` from the last `SMAOracle.lastUpdate`.

b. **Details:** `PoolKeeper` can keep different pools. All pools kept by the same keeper share the same observer defined on the `PoolKeeper` contract. If the keeper is using a `SMAOracle` as the observer writer then `performUpkeepSinglePool` may revert the second time it is called because `SMAOrcale.poll` will revert with "*SMA: Too early to update.*"

c. **Github Permalinks:** `observer.poll`

d. **Mitigation:** Ensure that if the keeper has more than one pool to keep updated and it is using a `SMAOracle` as the observer writer, then the poll method will be called only after at least `SMAOracle.updateInterval` from the last `SMAOracle.lastUpdate`.

53. `PoolToken` mint and burn functions always return true

a. **Summary:** `PoolToken` mint and burn functions always returning true

deviates from standard ERC20 behaviour.

b. **Details:** Standard ERC20 `_mint` and `_burn` functions do not return boolean but just revert on failure. However, `PoolToken` mint and burn always return true which is confusing and not expected even by the callers of those functions. `LeveragedPool.mintTokens` and `LeveragedPool.burnTokens` have a `require` on calls to these functions expecting them to sometimes return false.

c. **Github Permalinks:** `mint`, `burn`, `mintTokens`, `burnTokens`

d. **Mitigation:** Update both functions removing the `return true` statements and update `LeveragedPool.mintTokens` and `LeveragedPool.burnTokens` to remove the `require` statements.

54. Follow best-practices for smart contract layout

a. **Summary**: The recommended layout order for a smart contract is state variables, events, modifiers, constructor and functions.

b. **Details**: In `AutoClaim`, `LeveragedPool` and `PoolCommitter`, the modifier is placed at the end. This deviates from best-practice and affects readability.

c. **Github Permalinks**: `AutoClaim`, `LeveragedPool`, `PoolCommitter`

d. **Mitigation**: Update contract layout to recommended best-practice.

55. Incorrect value emitted in event

a. **Summary:** `PaidClaimRequestUpdate` event's `newReward` is not accurate when reward is updated.

b. **Details:** When reward is updated, the event `PaidClaimRequestUpdate` emits the increment value `msg.value` instead of the total updated reward value `request.reward`.

c. **Github Permalinks:** `PaidClaimRequestUpdate`

d. **Mitigation:** Use `request.reward` instead of `msg.value`.

56. Inconsistent function return patterns affects readability

a. **Summary**: Most functions explicitly return a value while few others assign a value to a named return variable.

b. **Details**: If many functions explicitly return a value, one may incorrectly infer that functions assigning a value to a named return variable are not actually returning anything.

c. **Github Permalinks**: `getPrice`, `getPriceAndMetadata`

d. **Mitigation**: Consider refactoring functions to standardise on one return pattern preferably the explicit return.

57. Functions with duplicated logic

a. **Summary:** Functions `getPrice` and `poll` implement the same logic in `ChainlinkOracleWrapper`.

b. **Details:** Both functions `getPrice` and `poll` implement a call to `_latestRoundData` and return the price.

c. **Github Permalinks:** `getPrice`, `poll`

d. **Mitigation:** One of them can be removed to simplify logic and save gas.

58. Storage packing optimisation

a. **Summary:** Optimise storage packing by reorganising variables declarations.

b. **Details:** State variable declaration order and their sizes are used to determine their storage slots. Reorganising declaration order can lead to more efficient packing and usage of fewer storage slots leading to gas savings. Note that packing may lead to some additional overhead due to masking/unmasking of shared slot variables during their SLOAD/SSTORE depending on their access patterns, which has to be factored in.

c. **Github Permalinks:** `LeveragedPool`

d. **Mitigation:** Evaluate types and declaration order of state variables in the context of storage packing, number of storage slots and gas efficiency.

59. Potential transfer of 0 tokens

a. **Summary**: A `secondaryFeeSplitPercent` of 100 would cause a transfer of 0 tokens to `feeAddress`.

b. **Details**: It is possible for `secondaryFeeSplitPercent` to be 100 which would cause a transfer of 0 tokens to `feeAddress`. This can be checked to avoid an unnecessary transfer.

c. **Github Permalinks**: `feeAddress`

d. **Mitigation**: Add a 0 check before transfer to `feeAddress`.

60. Naming and description of functions and variables

a. **Summary:** Names and descriptions of functions and variables should be distinct and clearly indicate their purpose and effects to improve code readability, development and maintenance. They should also follow best-practices and be consistent when it comes to naming convention.

b. **Details:**

i. **Unexpected side-effect in `AutoClaim.makePaidClaimRequest`:** `AutoClaim.makePaidClaimRequest` will claim a previously pending and claimable request before creating the requested claim-request which is not indicated by the function name or description.

ii. `updateInterval` and `updateIntervalId` variable names are very similar.

iii. `PoolKeeper.checkUpkeepSinglePool` (and its multiple counterpart) may be renamed to `PoolKeeper.isUpkeepRequiredSinglePool` to better indicate implemented functionality.

iv. `PoolKeeper.fixedPoint` constant is not declared under the *Constants* heading, is not uppercased, and does not explicitly specify

its visibility.

v. `updateBalanceSingleCommitment` is a view function that cannot update anything.

c. **Github Permalinks:** `AutoClaim.makePaidClaimRequest`, `updateInterval and updateIntervalId`, `PoolKeeper.checkUpkeepSinglePool`, `PoolKeeper.fixedPoint`, `updateBalanceSingleCommitment`

d. **Mitigation:** Use distinct/clear names and descriptions for functions and variables. Consider following best-practices and be consistent when it comes to naming convention.

61. Error messages

a. **Summary**: Custom errors may reduce gas costs.

b. **Details**: Throughout the project, `require` conditional checks use a custom error message. To save gas, this can be replaced by custom errors which were introduced in Solidity `0.8.4`.

c. **Github Permalinks**: `LeveragedPool`

d. **Mitigation**: Consider using custom errors instead of `require` checks.

62. Deployment of malicious pools

a. **Summary:** Given that deployment of pools is permissionless, one can deploy a malicious pool with arbitrary tokens/functionality that steal users' collateral.

b. **Details:** When deploying a new pool, the deployer has control over parameters such as `quoteToken`, `oracleWrapper`, `settlementEthOracle`, `invariantCheckContract`, `poolKeeper`, `frontRunningInterval`, `updateInterval`, `fee` and `leverageAmount` to different extents. Deployers can therefore deploy pools with functionality enabled to steal the collateral deposited in obvious or hidden ways. For e.g., an `oracleWrapper` where they control the price, an ERC20 token with incorrect accounting/logic, a token with hooks/callbacks (ERC777/ERC1155) which allows reentrancy or an invariant contract with unexpected functionality that can lead to loss/lock of user collateral.

c. **Github Permalinks:** `deployPool`

d. **Mitigation:** Evaluate appropriate checks and measures to mitigate risks from malicious pools.

63. DoS of pool upkeep

a. **Summary:** Gas price computation for keeper rewards may not ensure upkeeping of the pool.

b. **Details:** Reward for the keeper is computed by compensating for gas along with additional tip for incentives. The `gasPrice` however is set to a default value of 10 gwei with the `PoolKeeper` owner expected to change this using

`setGasPrice` depending on fluctuating gas prices. As noted in its `@dev` comment: "*This function is only necessary due to the L2 deployment of Pools -- in reality, it should be `BASEFEE.`*" It is however not clear that this gas price computation process is effective at maintaining correct gas prices at all times and may lead to fluctuating keeper incentives that prevent expected real-time upkeeping of pools. This could lead to a DoS of pool upkeep.

c. **Github Permalinks:** `gasPrice`, `payKeeper`, `setGasPrice`

d. **Mitigation:** Reevaluate gas price computation to account for EIP-1559 with `BASEFEE` and priority fee considered to maintain a dynamic peg to real-time gas prices.

64. Accounting for only two update intervals may pause protocol

a. **Summary:** `InvariantCheck` accounting for only two update intervals may lead to protocol pause because of miscalculations.

b. **Details:** Protocol allows one to create pools with `frontRunning` intervals longer than the `updateInterval` period. Without any threshold checks on the two intervals, commits may be scheduled further out than 2 periods. `InvariantCheck` calls `getPendingCommits` which only returns commits for the next 2 update intervals instead of returning all future intervals. This mismatch between what is allowed and checked may lead to incorrect pool accounting that triggers a pause.

c. **Github Permalinks:** `deployPool`, `checkInvariants`, `getPendingCommits`

d. **Mitigation:** `getPendingCommits` needs to retrieve all future commits and not just the next 2.

CARE: Disclaimer

CARE is *not* an audit as is typically performed by smart contract security audit firms. CARE participants aim primarily to identify commonly known security pitfalls & best-practices but *not* necessarily application-specific or economic vulnerabilities which are expected to be the focus of future security audits. CARE assumes (as notified and agreed upon earlier in the CARE SoW) that the project will get one or more security audits *after* CARE to cover those aspects. Furthermore, CARE is a best-effort endeavour. Secureum will not be held responsible for any loss/lock of funds/services resulting from vulnerabilities/exploits in projects after they have gone through CARE review.