Tracer DAO

Perpetual Pools

Smart Contract Security Assessment

Version: 1.1

Contents

Introduction

Sigma Prime was commercially engaged to perform a time-boxed security review of the Perpetual Pool smart contracts. The review focused solely on the security aspects of the Solidity implementation of the contract, though general recommendations and informational comments are also provided.

Disclaimer

Sigma Prime makes all effort but holds no responsibility for the findings of this security review. Sigma Prime does not provide any guarantees relating to the function of the smart contract. Sigma Prime makes no judgements on, or provides any security review, regarding the underlying business model or the individuals involved in the project.

Document Structure

The first section provides an overview of the functionality of the Perpetual Pool smart contracts contained within the scope of the security review. A summary followed by a detailed review of the discovered vulnerabilities is then given which assigns each vulnerability a severity rating (see Vulnerability Severity Classification), an *open/closed/resolved* status and a recommendation. Additionally, findings which do not have direct security implications (but are potentially of interest) are marked as *informational*.

Outputs of automated testing that were developed during this assessment are also included for reference (in the Appendix: Test Suite).

The appendix provides additional documentation, including the severity matrix used to classify vulnerabilities within the Perpetual Pool smart contracts.

Overview

TracerDAO's perpetual pool smart contracts enable users to take leveraged long and short positions on any asset without fear of liquidation. Market creators can deploy leveraged pools via the factory contract, generating `ERC20` long and short pool tokens in the process. Once a pool has been created, users can mint and burn tokens by committing to `LongMint`, `LongBurn`, `ShortMint` or `ShortBurn` pool tokens. These actions are limited to a regular interval consisting of an update interval and a frontrunning interval, the former being the allotted slot in each interval whereby users can `commit/uncommit` and have their actions included in the next pool update.

Rebalancing events occur on the boundary of this regular interval, facilitating the transfer of value from the losing side to the winning side in proportion to the pool's leverage. Chainlink Oracles enable leveraged pools to gain access to off–chain pricing data of the assets being tracked. By taking advantage of Chainlink's extensive oracle network, TracerDAO is able to preserve a decentralized ethos.

Interactions by external actors can be simplified into the following:

- A user `commits` funds to mint pool tokens and takes a long or short position in the leveraged pool.
- A user `commits` to burn their pool tokens and exit their long or short position in the leveraged pool. Users receive their share of the long or short pool balance.

- A user actions to `uncommit` a previously committed action and have their tokens returned if in their initial commit, they intended to `LongMint` or `ShortMint` pool tokens.
- Pool keepers are tasked with performing upkeeps on any deployed pools at a regular update interval. They are reimbursed for the cost incurred with an additional tip paid on-top of this. The tip consists of a 5% base tip and a 5% tip for each elapsed block since the pool was eligible for upkeep.
- Pool creators are able to deploy new leveraged pools with unique deployment parameters. Pools are created through the `PoolFactory.sol` contract. It is important to note that the pool creator does not have any ownership/governance privileges for the underlying leveraged pool, however, they do have control over the configured oracles used to query price data for the `quoteToken` and `settlementToken` (denominated in ETH) assets.

Security Assessment Summary

This review was conducted on the Perpetual Pool files hosted on an internal Mycelium repository and were assessed at commit 2ee499a.

Note: the OpenZeppelin libraries and dependencies were excluded from the scope of this assessment.

TracerDAO has taken an optimistic approach to oracle security and as such, assumptions have been made about the reliable selection of oracles by market creators. TCR-05 references the related security assumption and has been marked as *"informational"* to reflect TracerDAO's threat model.

The manual code review section of the report, focused on identifying any and all issues/vulnerabilities associated with the business logic implementation of the contracts. Specifically, their internal interactions, intended functionality and correct implementation with respect to the underlying functionality of the Ethereum Virtual Machine (for example, verifying correct storage/memory layout). Additionally, the manual review process focused on all known Solidity anti-patterns and attack vectors. These include, but are not limited to, the following vectors: re-entrancy, front-running, integer overflow/underflow and correct visibility specifiers. For a more thorough, but non-exhaustive list of examined vectors, see [**?**, **?**].

To support this review, the testing team used the following automated testing tools:

- Mythril: `https://github.com/ConsenSys/mythril`
- Slither: `https://github.com/trailofbits/slither`
- Surya: `https://github.com/ConsenSys/surya`

Output for these automated tools is available upon request.

Findings Summary

The testing team identified a total of 12 issues during this assessment. Categorized by their severity:

- Critical: 1 issue.
- Medium: 1 issue.
- Low: 2 issues.
- Informational: 8 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Perpetual Pool smart contracts included in the scope of this review. Each vulnerability has a severity classification which is determined from the likelihood and impact of each issue by the matrix given in the Appendix: Vulnerability Severity Classification.

A number of additional properties of the contracts, including gas optimisations, are also described in this section and are labelled as "informational".

Each vulnerability is also assigned a **status**:

- ***Open:*** the issue has not been addressed by the project team.
- ***Resolved:*** the issue was acknowledged by the project team and updates to the affected contract(s) have been made to mitigate the related risk.
- ***Closed:*** the issue was acknowledged by the project team but no further actions have been taken.

Summary of Findings

ID	Description	Severity	Status
TCR-01	Denial-of-Service Attack on Leveraged Pool via Multiple Commits	Critical	Open
TCR-02	User Can Reset `earliestCommitUnexecuted` State Variable	Medium	Open
TCR-03	Invalid `string` to `bytes16` Conversion	Low	Open
TCR-04	Divide Before Multiply	Low	Open
TCR-05	Ownable Oracle Allowed in `ChainlinkOracleWrapper`	Informational	Open
TCR-06	Unnecessary Multiplication by `fixedPoint`	Informational	Open
TCR-07	Potential MEV Attack When Users Commit	Informational	Open
TCR-08	`onlyGov` Role Can Withdraw Users' Funds	Informational	Open
TCR-09	Known Issues	Informational	Open
TCR-10	Test Coverage Improvements	Informational	Open
TCR-11	Gas Optimisations	Informational	Open
TCR-12	Miscellaneous Tracer Contract Issues	Informational	Open

TCR-01	Denial-of-Service Attack on Leveraged Pool via Multiple Commits		
Asset	PoolCommitter.sol		
Status	Open		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

Due to the iterative nature of how commits are executed, a malicious user is able to dilute the queue of pending commits and effectively prevent a pool keeper from performing its duties. This is made possible by the lack of a minimum commit amount and an unbounded commit queue.

The `PoolCommitter.sol` contract contains the functionality to enable users to interact with their leveraged pool of choice by depositing and withdrawing their tokens. To do this, users will commit to an action by taking a position consisting of one of the following types; `LongMint`, `LongBurn`, `ShortMint` or `ShortBurn`.

The `commit()` function checks that the amount to be committed is a positive amount and proceeds to add the commit to a queue of pending commits. This check is found on line [40] of `PoolCommitter.sol`:

```
require(amount > 0, "Amount must not be zero");
```

Pending commits are executed by the pool keeper on each eligible update interval by use of the `PoolKeeper.performUpkeepSinglePool()` function. It is possible for a malicious user to create multiple commits containing small amounts that satisfy the aforementioned check, thereby diluting the queue of pending commits. As a result, a pool keeper may be unable to perform an upkeep within the given block gas limit, ergo preventing all pool deposits and withdrawals.

Recommendations

Consider enforcing an implementation of these two strategies:

- A minimum commit size for all deposits and withdrawals required by the `commit()` function.
- A maximum number of commits that can be executed in a given update interval. This can be applied by bounding the iterative `executeAllCommitments()` function to a maximum size, or by restricting the number of commits that can be made through the `commit()` function. Although, the latter option lacks frontrunning protection for users.

Note that the values for these two strategies need to be chosen such that the cost of a potential attack would be prohibitive.

TCR-02	User Can Reset `earliestCommitUnexecuted` State Variable		
Asset	`PoolCommitter.sol`		
Status	Open		
Rating	Severity: Medium	Impact: High	Likelihood: Low

Description

Users enter and exit their long or short positions by calling `commit()` with the relevant action. These commits are then placed into a queue where the earliest and latest commit IDs are tracked as state variables `earliestCommitUnexecuted` and `latestCommitUnexecuted` respectively. Through nefarious means, a user could put this mechanism in an unexpected state in which `earliestCommitUnexecuted` is reset to 0. If `commitIDCounter` is sufficiently high, this could potentially disable the pool keeper's ability to perform upkeeps due to an *"Out of Gas"* error.

Consider the following attack scenario:

- Alice creates 5 separate commits with IDs 1, 2, 3, 4 and 5.
- Alice then decides to uncommit commits with IDs 2, 3 and 4.
- In its current state, `earliestCommitUnexecuted` corresponds to Alice's commit of ID 1. Similarly, `latestCommitUnexecuted` corresponds to Alice's commit of ID 5.
- Alice then uncommits her original commit of ID 5, updating `latestCommitUnexecuted` to point to her commit with ID 4.
- Alice also decides to uncommit her original commit of ID 1, updating `earliestCommitUnexecuted` to point to her commit with ID 2.
- A nefarious user, Bob, sees this actions and decides he wants to abuse the `PoolCommitter.sol` smart contract.
- Bob decides to perform an upkeep on the same pool Alice has commits waiting to be executed by calling `PoolKeeper.performUpkeep()`. This resultingly calls `executeAllCommitments()` which iterates through all pending commits, skipping any previously uncommitted or non-existent commits.
- Because `latestCommitUnexecuted` points to an empty commit, the branch found on line [144] is reached, setting the local state variable `nextEarliestCommitUnexecuted` to `latestCommitUnexecuted + 1`.

```
if (_commit.owner == address(0)) {
    // Commit deleted (uncommitted) or already executed
    nextEarliestCommitUnexecuted += 1; // It makes sense to set the next unexecuted to
        the next number
    continue;
}
```

- After the upkeep is completed, `earliestCommitUnexecuted` is 5 and `latestCommitUnexecuted` is 4.
- If no one makes a commit before the next update interval and the process is repeated where Bob performs another upkeep. Not only are no commits executed (to be expected when there are no pending commits), but the `executeAllCommitments()` function sets `earliestCommitUnexecuted` to an undefined local state variable, i.e. to 0.

Recommendations

Consider updating the logic of `_uncommit()` such that this logic bug is avoided. A potential simpler solution could be to remove the local variable `nextEarliestCommitUnexecuted` all together and instead perform another check at the end of the function to ensure we reached the last commit. The check could be a duplicate of the following:

```
if (i == latestCommitUnexecuted) {
    // We have reached the last one
    earliestCommitUnexecuted = NO_COMMITS_REMAINING;
    return;
}
```

Fortunately, commits are deleted after being executed, so resetting `earliestCommitUnexecuted` does not pose any concern for double minting/burning tokens. However, this could add additional gas costs to the keeper for which they are compensated for. Therefore, keepers are incentivized to also abuse this bug in order to earn increased payouts for their upkeeps.

TCR-03	Invalid `string` to `bytes16` Conversion		
Asset	`PoolSwapLibrary.sol`		
Status	Open		
Rating	Severity: Low	Impact: Medium	Likelihood: Low

Description

The function `getAmountOut()` is used to calculate the amount of tokens a user is entitled to based on the current `ratio` and `amountIn`.

The library `ABDKMathQuad` has two representations for the number zero, being positive and negative zero.

There is a check seen in the following code snippet showing the check for both positive and negative zero.

```
if (ABDKMathQuad.cmp(ratio, 0) == 0 || ABDKMathQuad.cmp(ratio, bytes16("0x1")) == 0) {
```

There are two issues with the second condition which is the check for negative zero. First, solidity conversion from `string` to `bytes16` will treat the string as an ASCII array. Hence `bytes16("0x1")` will be treated as the array `[30, 78, 31]`.

The second issue is that negative zero has the first bit set to zero rather than the first byte. That is negative zero has the first byte as `0x80 = 0b1000_0000` not `0x01 = 0b0000_0001`.

As a result, the `getAmountOut()` function will not return `amountIn` for negative zero. To exploit this issue a user would need to call this function when the ratio is about $\frac{1}{2^{4087}}$ which should not be reachable by the protocol in normal conditions.

Recommendations

This issue may be mitigated by using constants for positive and negative zero. For example

```
bytes16 internal constant POSITIVE_ZERO = 0x00000000000000000000000000000000;
bytes16 internal constant NEGATIVE_ZERO = 0x80000000000000000000000000000000;
```

TCR-04	Divide Before Multiply		
Asset	PoolKeeper.sol and PoolCommitter.sol		
Status	Open		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Solidity is unable to handle floating point numbers and will therefore truncate integers during division. This may result in an unnecessary loss of precision when attempting to execute commitments for minting and burning. Due to the high precision of `ether` values, this issue is of low severity. The following functions are affected by this issue:

- `mintAmount` in `PoolSwapLibrary.getMintAmount()`;
- `amountOut` in `PoolSwapLibrary.getAmountOut()`;
- `keeperReward` in `PoolKeeper.keeperReward()`.

Burn Commitments:

After `LeveragedPool.executePriceChange(oldPrice, newPrice)` has executed and long and short balances have been updated, certain price inputs create changes to long and short balances that are not favourable for burning commitments. For example, assuming an initial long and short balance of `100`, `LeveragedPool.executePriceChange(1000, 3000)` produces a long balance of `166.66666666666669` and short balance of `33.33333333333333`. Both these balances are used when burning to calculate the `amountOut`, and due to division occuring before multiplication we are left with a missing wei in the leveraged pool that cannot be claimed due to rounding.

Mint Commitments:

When minting, the `PoolSwapLibrary.getMintAmount()` relies on multiplication of `amountIn * ratio`, where the ratio is `(tokenSupply + inverseShadowBalance) / balance`. Again a similiar imbalance can be obtained by producing interesting `inverseShadowBalance` to `balance` ratios that lead to floating points. When large floating points occur, losses of a single wei for those mintings may follow.

Keeper Rewards:

The `keeperReward()` function in the `PoolKeeper.sol` contract also contains another division before multiplication.

This can be seen on line [183] and with the result of `_tipPercent`, a division operation, then being used in a subsequent multiplication. Due to the use of fixed points, the impact of this issue is more limited than the previous occurences described above.

Recommendations

The testing team recommends performing all multiplication operations before any division whenever possible.

TCR-05	Ownable Oracle Allowed in `ChainlinkOracleWrapper`
Asset	`ChainlinkOracleWrapper.sol`
Status	**Open**
Rating	Informational

Description

The `ChainlinkOracleWrapper` contract is a wrapper implementation for fetching price feed data, typically from the Chainlink network. Market creators deploy an oracle wrapper contract before deploying a leveraged pool through `PoolFactory.deployPool()`. The wrapper contract contains its own `onlyOwner` role which is delegated to the contract deployer. In most cases, the market creator and the deployer of the oracle wrapper contract will be the same.

This opens up the potential for a third–party market creator to intentionally manipulate the price of a pool's asset by changing the underlying oracle price feed. If users have already taken long and short positions in the leveraged pool, the market creator could take a large position on one side of the pool, manipulate the price and drain the balance of the pool on the other side.

Recommendations

Ensure that this is understood by TracerDAO's perpetual pool users. While the testing team understands that initial V1 deployment involves TracerDAO deploying a number of reputable pools using Chainlink oracles as price feeds, users may be unaware of any potential dangers when using markets deployed by third–party entities.

TCR-06	Unnecessary Multiplication by `fixedPoint`
Asset	`PoolKeeper.sol`
Status	Open
Rating	Informational

Description

The map variable `executionPrice` stores the current execution price as an `int256`, which is fetched from the `ChainlinkOracleWrapper.sol` contract. The price returned by `ChainlinkOracleWrapper` is in WAD (10^{18}) units.

```
int256 latestPrice = IOracleWrapper(pool.oracleWrapper()).getPrice();
// Start a new round
int256 lastExecutionPrice = executionPrice[_pool];
executionPrice[_pool] = ABDKMathQuad.toInt(
    ABDKMathQuad.mul(ABDKMathQuad.fromInt(latestPrice), fixedPoint));
```

The snippet above from `performUpkeepSinglePool()` shows the price from `ChainlinkOracleWrapper` is futher multiplied by `fixedPoint` which is $1 * 10^{18}$, thereby giving `executionPrice` units of 10^{36}.

The price is only used as a ratio, where we have $\frac{latestPrice}{lastExecutionPrice}$ or the inverse. Thus the additional units will cancel out.

However, if the price overflows the 112 bit significand ($5 * 10^{33}$) some precision will be lost.

A similar issue can be seen in the function `keeperReward()`, where `_tipPercent` is unnecessarily multiplied by `fixedPoint`. Since `_tipPercent` is only stored as a quadruple-precision number and not converted to an integer, it is unnecessary to multiply the value by `fixedPoint`. This is because quadruple-precision representation can already adequately handle the ratios potentially less than one (1) and will not gain additional precision from being a larger value.

```
bytes16 _tipPercent = ABDKMathQuad.mul(
    ABDKMathQuad.fromUInt(keeperTip(_savedPreviousUpdatedTimestamp, _poolInterval)),
    fixedPoint
);
// amount of settlement tokens to give to the keeper
_tipPercent = ABDKMathQuad.div(_tipPercent, ABDKMathQuad.fromUInt(100));
int256 wadRewardValue = ABDKMathQuad.toInt(
    ABDKMathQuad.add(
        ABDKMathQuad.fromUInt(_keeperGas),
        ABDKMathQuad.div((ABDKMathQuad.mul(
            ABDKMathQuad.fromUInt(_keeperGas), _tipPercent)), fixedPoint)
    )
);
```

Recommendations

Consider removing the multiplication by `fixedPoint` for `executionPrice`. Additionally, consider removing both the multiplication and division of `fixedPoint` from `_tipPercent` and `wadRewardValue`.

TCR-07	Potential MEV Attack When Users Commit
Asset	`PoolCommitter.sol`
Status	**Open**
Rating	Informational

Description

Pool keepers play a key role in asserting regular price updates are made to a leveraged pool following the execution of any pending commits. A frontrunning interval exists within the regular update interval such that any commitments made during this frontrunning interval are not executed until the following upkeep.

Miners are able to manipulate `block.timestamp` as long as it satisfies the following properties:

- `block.timestamp` is greater than the parent block's timestamp.
- `block.timestamp` is within a 15 second time variance of it's parent's `block.timestamp`.

Users intending to enter the pool by minting long or short pool tokens are incentivized to make this decision just before the frontrunning interval starts. This allows them to utilise the latest price data for the asset being tracked by the pool. As a result, it is possible for miners to effectively censor users by increasing `block.timestamp` such that `executeAllCommitments()` skips certain commits and `uncommit()` reverts for other commits.

This attack is not entirely viable unless there is a drastic short term price movement, resulting in a lagging SMA that is likely to follow the same trend but at a reduced velocity. Therefore, a malicious user could pay a miner to censor certain transactions and have them pushed to the following update interval. As `commit()` transactions are likely to still succeed, miners are able to perform an MEV attack without losing out on any fees that they would normally generate. A scenario where this would be likely could involve a miner censoring any commits related to users on the losing side of the pool. This would effectively delay their commits to exit their long or short positions until the following update interval, potentially generating increased profits for the winning side.

Recommendations

Ensure that users are aware of MEV threats and advise them to avoid using the `commit()` and `uncommit()` functions when `block.timestamp` is close to the frontrunning interval.

TCR-08	`onlyGov` Role Can Withdraw Users' Funds
Asset	`LeveragedPool.sol`
Status	**Open**
Rating	Informational

Description

The `onlyGov` role is configured to be the owner of the `PoolFactory.sol` contract. This gives permissions to update various settings in a leveraged pool. A compromised `onlyGov` role is able to extract funds from a leveraged pool by firstly setting the `keeper` address to an account in their control using the `setKeeper()` function. They are subsequently able to call `payKeeperFromBalances()` and withdraw any arbitrary amount from the leveraged pool.

Recommendations

Ensure that the associated risks are clear and well understood. While the testing team understands that the `onlyGov` role is highly unlikely to be compromised, mitigating any issues related to this event ensures users' funds continue to be protected.

TCR-09	Known Issues
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

The testing team has identified four (4) issues that have been previously flagged by the TracerDAO team. These include:

- `Safe Governance Transfer`
- `Keeper Rewards`
- `Keeper Gas Price Usage`
- `New Keeper Propagation`

Safe Governance

The current transfer pattern of governance involves calling the function `LeveragedPool.transferGovernance()` which instantly updates the `onlyGov` role. This allows the current `onlyGov` account to set any arbitrary address, excluding the zero address.

If the address is entered incorrectly or set to an unknown address, the `onlyGov` role is lost forever, preventing several important `onlyGov` functions from being called. Those functions include `LeveragedPool.setKeeper()` and `LeveragedPool.updateFeeAddress()`.

Recommendations

This scenario is typically mitigated by implementing a two stage `transferOwnership` pattern. A new governance address is selected, then the selected address must call a `claimOwnership()` before the owner is changed. This ensures the new owner address is accessible.

Keeper Rewards

Pool keepers are potentially not rewarded for performing an upkeep on a pool if the balance of the pool is insufficiently funded. This may be due to the fact that the pool has recently been deployed or has become inactive overtime.

Currently, there is no way for keepers to check if their call to upkeep a pool will successfully payout a reward without simulating a JSON-RPC `eth_call`. This adds additional complexity to a keeper's role and should be avoided.

Recommendations

Consider implementing a view function to enable pool keepers to check a pool's balance before proceeding with a pool upkeep.

Keeper Gas Price Usage

The `PoolKeeper.sol` contract currently uses a fixed gas pricing model when calculating the amount to pay a keeper for performing an upkeep. There is a leftover `TODO` in `PoolKeeper.performUpkeepSinglePool()` line [`109`] which may result in transactions failing in a mainnet environment.

Recommendations

Consider utilising a gas price oracle or wait for an EIP–1559 compliant solution in solidity.

New Keeper Propagation

The `PoolFactory.setPoolKeeper()` function does not propagate to all subsequent pools, making keeper updates complex and costly to perform.

Recommendations

Consider updating the `PoolFactory.setPoolKeeper()` function to propagate a keeper update to all valid pools in the contract.

TCR-10	Test Coverage Improvements
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

Adequate test coverage and regular reporting is an essential process to ensuring the codebase works as intended. Insufficient code coverage may lead to unexpected issues and regressions arising due to changes in the underlying smart contract implementation.

Recommendations

Consider updating the tests for TracerDAO's smart contract suite to include interactions with all code statements and branches.

Incorporate regular test coverage reporting into the development workflow. solidity-coverage supports Hardhat as of v0.7.12.[1]

[1] https://github.com/sc-forks/solidity-coverage#hardhat

TCR-11	Gas Optimisations
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

1. `PoolCommitter.sol` may cache the addresses for `poolTokens` so the addresses don't need to be fetched from `LeveragedPool.sol` each time, saving gas on external calls.

2. `PoolCommitter.commitTypeToUint()` may remove the `ShortMint` `if else` branch since the default `else` is zero (0).

3. `PoolKeeper.performUpkeepSinglePool()` makes three separate calls to `LeveragedPool.sol`, each external call has gas overhead. Consider having a single function which has a tuple of three values as the return. The external calls are `oracleWrapper()`, `updateInterval()` and `lastPriceTimestamp()`.

4. The number of math operations in `PoolSwapLibrary.getLossMultiplier()` can be reduced to save gas and reduce code complexity. Consider the following code snippet:

```
ABDKMathQuad.pow_2(
        ABDKMathQuad.mul(
            leverage,
            ABDKMathQuad.log_2(
                ABDKMathQuad.add(
                    ABDKMathQuad.mul(direction < 0 ? one : zero, ratio),
                    ABDKMathQuad.div(
                        ABDKMathQuad.mul(direction >= 0 ? one : zero, one), ratio)
                )
            )
        )
    );
```

   The code can be simplified by moving the ternary operator and removing the multiplications by `one`. Consider the following code which produces the same output.

```
ABDKMathQuad.pow_2(
        ABDKMathQuad.mul(
            leverage,
            ABDKMathQuad.log_2(
                direction < 0 ? ratio : ABDKMathQuad.div(one, ratio)
            )
        )
    );
```

5. The function `PoolSwapLibrary.calculatePriceChange()` can have the calculations for `totalFeeAmount` simplified.

```
uint256 totalFeeAmount = 0;

// fee is enforced to be < 1.
// Therefore, shortFeeAmount < shortBalance, and longFeeAmount < longBalance
shortBalance = shortBalance - shortFeeAmount;
totalFeeAmount = totalFeeAmount + shortFeeAmount;
longBalance = longBalance - longFeeAmount;
totalFeeAmount = totalFeeAmount + longFeeAmount;
```

The code calculations may be reduced to the following.

```
shortBalance = shortBalance - shortFeeAmount;
longBalance = longBalance - longFeeAmount;
uint256 totalFeeAmount = shortFeeAmount + longFeeAmount
```

6. Gas savings by utilising solc optimizer in hardhat config.
7. There are a number of functions which can be declared `external`, generating gas savings upon contract deployment and on each function call.

   These functions are:

   - `PoolSwapLibrary.getBalancesAfterFees()`
   - `PoolSwapLibrary.calculatePriceChange()`
   - `PoolCommitter.getCommit()`

   Consider changing the above functions from `public` to `external`.

8. `LeveragedPool.getOraclePrice()` can be used to consolidate a number of functions in `PoolKeeper.sol`, namely; `newPool()`, `performUpkeepSinglePool()` and `keeperGas()`. This helps to minimise the number of state reads, saving gas on each function call. If this is intended behaviour, then `LeveragedPool.getOraclePrice()` can have its visibility updated from `public` to `external`.
9. `pairTokenBase` and `poolBase` addresses in `PoolFactory.sol` can be declared as `immutable` for potential gas savings as the variable is only read by the contract after being assigned in the `constructor()`.
10. Potential gas savings by avoiding updates to the leveraged pool's long and short balances in `LeveragedPool.executePriceChange()`. If there is no change in price, external calls made in this function can be safely avoided.
11. There are potential redundant `SLOAD` operations in `PoolCommitter.executeAllCommitments()` used to track the `nextEarliestCommitUnexecuted` state variable. `nextEarliestCommitUnexecuted` is always equal to `i` and can therefore be replaced with `i` in line [166].
12. `PoolCommitter.executeAllCommitments()` is performing additional `SLOAD` operatons in the `for` loop. It may be more gas efficient to load these into local variables, utilising the cheaper `MLOAD` opcode.

Recommendations

Gas optimisations are often a trade–off between cost and code simplicity. Review the gas optimisations provided and consider implementing them where appropriate.

TCR-12	Miscellaneous Tracer Contract Issues
Asset	`contracts/*`
Status	Open
Rating	Informational

Description

This section details miscellaneous findings discovered by the testing team that do not have direct security implications:

1. **Unnecessary Casting:**
   - `lastPriceTimestamp` in `LeveragedPool.sol` is stored as `uint256` and thus does not need to be cast to `uint40` before automatically being cast back to `uint256`.
     Noting that it is used in the event `IPoolCommitter.Commit` for the field `created` as `uint40`.
   - The line in `uint256 scaler = uint256(10**(MAX_DECIMALS - _decimals));` of `PoolSwapLibrary.fromWad()` does not need explicit casting to `uint256` as all numbers are already of type `uint256`.
   - `LeveragedPool.payKeeperFromBalances()` performs an unnecessary `uint256` cast of the `amount` input. Consider removing the `uint256` cast in line [117] of `LeveragedPool.sol`.
2. **Duplicate Modifiers:**
   - Modifiers `onlyKeeper` and `onlyPoolKeeper` perform the same functionality in `LeveragedPool.sol`. Therefore, one of these modifiers can be safely removed and all instances of its use can be replaced with the other modifier.
3. **Lack of SPDX Licence Identifier:**
   - The `IPoolToken.sol` interface contract is missing an SPDX identifier which correctly licenses the contract for open source development.
4. **List of Typos:**
   - At `PoolCommitter.sol` line [64], line [67], line [70] and line [261], there is a typo in the function comments and the `onlyFactory` revert message. *"commiter"* should be *"committer"* in the first three instances. The `onlyFactory` revert message should be updated to *"Committer: not factory"*
   - At `PoolFactory.sol` line [116] and line [117], there is a typo in the function comments. *"commiter"* should be *"committer"*.
   - `PoolCommitter.sol` and `LeveragedPool.sol` contracts contain inaccurate title comments. These should be updated to better reflect the seggregation of functionality between the two contracts.
   - By default `feeReceiver` in `PoolFactory.sol` is a public state variable, however, to be consistent the testing team recommends adding a specific `public` keyword to line [26].
   - The revert message in `LeveragedPool.initialize()` line [56-59] does not correctly reflect the fee check's behaviour. In its current implementation, a pool's fee cannot be initialized to 100%. This is likely intended behaviour, therefore it would be useful to update the revert message to better reflect this. Consider updating the revert message on line [56-59] from *"Fee is greater than 100%"* to *"Fee is greater than or equal to 100%"* or similar.

5. **Lack of Zero Address Validation:**

   - A number of contracts lack proper zero address validation and therefore may be initialized into an unexpected state. Ensure that contracts `PoolCommitter.sol`, `ChainlinkOracleWrapper.sol`, `PoolFactory.sol` and `PoolCommitterDeployer.sol` perform proper zero address validation in their constructors.

6. **Use of Default `CommitType`:**

   - If an invalid `CommitType` is selected when calling `PoolCommitter.commit()`, `PoolCommitter.commitTypeToUint()` will default to `ShortMint`. This may inhibit overall user experience and should be ideally avoided and replaced with a relevant revert message if an invalid `CommitType` is selected.

7. **Pause Mechanism:**

   - `LeveragedPool.sol` holds all assets related to open long and short positions and plays a key role in TracerDAO's perpetual pool system. Therefore, it may be useful to have a `Pausable` mechanism where the `onlyGov` role can trigger an emergency stop to all deposits and withdrawals. The Openzeppelin library contains an implementation of such a mechanism and can be applied by including the `whenNotPaused` and `whenPaused` modifiers to any target function.

8. **Potential Unexpected States:**

   - `PoolFactory.setMaxLeverage()` should not be settable to `0` as it puts the leveraged pool in a state where pools are no longer able to be deployed via `PoolFactory.deployPool()`. Ensure `maxLeverage` is never set to anything less than `1`.
   - If the initial oracle price queried in `PoolKeeper.newPool()` is `<= 0`, then keepers will be unable to perform their duties as the upkeep will fail at the following line in `LeveragedPool.executePriceChange()`.

     ```
     if (_oldPrice <= 0 || _newPrice <= 0) {emit PriceChangeError(_oldPrice, _newPrice);}
     ```

     Users can still mint and burn tokens as expected, but the market itself will become unusable. Consider adding a check in `PoolKeeper.newPool()` that ensures the starting price is a positive value.

9. **Unchecked ERC20 Return Value:**

   - `PoolCommitter.setQuoteAndPool()` makes an external `ERC20` call to `approve()` the pool address as a spender to the `PoolCommitter.sol` contract. Consider checking the return value of this external call.

10. **Keeper Frontrunning:**

    - The `PoolKeeper.performUpkeepSinglePool()` function is a public and unrestricted function allowing anyone to perform the duties of a keeper and be rewarded for their work. This helps to maintain a high availability network that is able to deal with a large number of leveraged pools. There is potential for keepers to frontrun each other as they compete for blockspace. Ensure this is understood by users wishing to interact with the `PoolKeeper.sol` contract.

11. **Unclear Keeper Payments:**

    - `LeveragedPool.payKeeperFromBalances()` utilises `PoolSwapLibrary.getBalancesAfterFees()` to determine the pool balances after the keeper's fees have been deducted. The following comment in `PoolSwapLibrary.sol` line [39] specifies that `reward` may be equal to the sum of the two pool balances:

      ```
      @dev Assumes shortBalance + longBalance >= reward
      ```

      Consider updating the following check in `payKeeperFromBalances()` or the aforcementioned `dev` comment in `PoolSwapLibrary.sol` to correctly match the intended behaviour.

      ```
      // If the rewards are more than the balances of the pool, the keeper does not get
             paid
      if (amount >= _shortBalance + _longBalance) {
          return false;
      }
      ```

Recommendations

Ensure that the comments are understood and acknowledged, and consider implementing the suggestions above.

Appendix A Test Suite

A non-exhaustive list of tests were constructed to aid this security review and are provided alongside this document. The `brownie` framework was used to perform these tests and the output is given below.

```
test_commit                                      PASSED   [3%]
test_uncommit                                    PASSED   [6%]
test_execute_all_commitments_rounding_error      XFAIL    [9%]
test_long_commit                                 PASSED   [12%]
test_short_commit                                PASSED   [16%]
test_dilute_commits                              PASSED   [19%]
test_execute_commitments_logic_bug               PASSED   [22%]
test_commit_dos                                  PASSED   [25%]
test_mint_token_imbalance                        XFAIL    [29%]
test_divison[1-3-0]                              PASSED   [32%]
test_divison[2-3-0]                              PASSED   [35%]
test_divison[10-11-0]                            PASSED   [38%]
test_divison[4-3-1]                              PASSED   [41%]
test_divison[7-6-1]                              PASSED   [45%]
test_divison[8-3-2]                              PASSED   [48%]
test_multiplication[1-3-2-0]                     PASSED   [51%]
test_multiplication[2-3-4-2]                     PASSED   [54%]
test_multiplication[10-11-4-3]                   PASSED   [58%]
test_multiplication[4-3-2-2]                     PASSED   [61%]
test_multiplication[7-6-6-6]                     PASSED   [64%]
test_multiplication[8-3-4-10]                    PASSED   [67%]
test_get_amount_out                              XFAIL    [70%]
test_mint                                        PASSED   [74%]
test_burn                                        PASSED   [77%]
test_deploy_pool_token                           PASSED   [80%]
test_deploy_pool_factory                         PASSED   [83%]
test_deploy_pool_committer_deployer              PASSED   [87%]
test_deploy_pool_committer                       PASSED   [90%]
test_deploy_pool_keeper                          PASSED   [93%]
test_deploy_chainlink_oracle_wrapper             PASSED   [96%]
test_deploy_leveraged_pool                       PASSED   [100%]
```

Appendix B Vulnerability Severity Classification

This security review classifies vulnerabilities based on their potential impact and likelihood of occurance. The total severity of a vulnerability is derived from these two metrics based on the following matrix.

Impact \ Likelihood	Low	Medium	High
High	Medium	High	Critical
Medium	Low	Medium	High
Low	Low	Low	Medium

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.

References

[1] Sigma Prime. Solidity Security. Blog, 2018, Available: https://blog.sigmaprime.io/solidity-security.html. [Accessed 2018].

[2] NCC Group. DASP - Top 10. Website, 2018, Available: http://www.dasp.co/. [Accessed 2018].

σ'