Tracer DAO

Perpetual Pools Updates

Smart Contract Security Assessment Report

Version: 2.0

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`, `ShortBurn`, `LongBurnShortMint` or `ShortBurnLongMint` 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` 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 `commits` to flipping their commit from short to long or vice-versa. They can perform this action on a single update interval.
- Users can incentivise claims by funding the `AutoClaim.sol` contract, ensuring their tokens are available for use upon execution of their commitment.
- Commitments can be aggregated to limit the use of conditional loops. A user's aggregated balance can be utilised in future commits, reducing the number of actions required to perform commits.
- 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 underlying token and `settlementToken` (denominated in ETH) assets.

Security Assessment Summary

This review was conducted on the files hosted on the Tracer repository and were assessed at commit 05dee8f.

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

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 23 issues during this assessment. Categorised by their severity:

- Critical: 1 issue.
- High: 4 issues.
- Medium: 4 issues.
- Low: 6 issues.
- Informational: 8 issues.

Detailed Findings

This section provides a detailed description of the vulnerabilities identified within the Perpetual Pool smart contracts. 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
TPP-01	Attempts to Long Mint From Aggregate Balance Will Pay Twice or Revert	Critical	Resolved
TPP-02	Keeper Rewards Are Not Scaled Down to Settlement Token Decimals	High	Resolved
TPP-03	Protocol Fees Can be Double Claimed	High	Resolved
TPP-04	Flip Commitments Are Not Charged Their Respective Mint And Burn Fee	High	Resolved
TPP-05	Pool Keepers Can Re-Enter `poolUpkeep()` if Tokens With Callbacks Are Used	High	Resolved
TPP-06	Stale Update Intervals Will Receive Burn Fees From Recent Update Intervals	Medium	Open
TPP-07	Secondary Fee Address Can Deny Pool Upkeeps by Reverting Upon Callback	Medium	Resolved
TPP-08	Update Timestamps For `SMAOracle` And `LeveragedPool` May Diverge Due to a Backlog of Update Intervals	Medium	Open
TPP-09	Change Interval Can be Arbitrarily Set by The Fee Controller to Brick The Leveraged Pool	Medium	Resolved
TPP-10	Arithmetic Operation May Unintentionally Revert When Calculating Keeper Reward	Low	Resolved
TPP-11	Invariant Checking Does Not Enforce a Paused `PoolCommitter` Contract	Low	Resolved
TPP-12	Keepers Will be Underpaid in a Post-Merge Ethereum Environment	Low	Closed
TPP-13	Unsafe Casting of `settlementTokenPrice`	Low	Resolved
TPP-14	`payKeeper()` Does Not Properly Handle Errors	Low	Resolved
TPP-15	First Committer of a New Pool Can Avoid Paying Fees	Low	Closed
TPP-16	Event Emission Always Records Zero Quantity	Informational	Resolved
TPP-17	Rewards Paid to Keeper Never Emitted in Events	Informational	Resolved
TPP-18	There is Currently no Incentive to Continuously Monitor Pools And Perform Invariant Checking	Informational	Resolved
TPP-19	Inconsistent Implementation of Fees	Informational	Closed
TPP-20	`newPool()` Does Not Handle Failed Polls	Informational	Closed
TPP-21	Upkeeps May be Performed on Stale Oracle Data	Informational	Closed
TPP-22	Miscellaneous `Tracer Perpetual Pools V2` General Comments	Informational	Resolved

TPP-23	Miscellaneous Gas Optimisations	Informational	**Resolved**

TPP-01	Attempts to Long Mint From Aggregate Balance Will Pay Twice or Revert		
Asset	`implementation/PoolCommitter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Critical	Impact: High	Likelihood: High

Description

When a user wants to mint pool tokens, they have the option to pay using `userAggregateBalance.settlementToken`. This is a record of settlement tokens owed to the user as a result of previous position burns, importantly it is intended such that you can mint a new position using this owed balance rather than withdrawing them then sending these settlement tokens back to the contract to mint.

The `commit()` function includes different pathways dictating the logic flow. These depend on how the user intends to pay for a new long mint and are broken due to a misplaced bracket on line [**316**]. As a result, when a user attempts to `long mint` using their aggregate balance as the source of funds, they also trigger a `long mint` funded via their account's token balance. If the user still has the contract approved to transfer the settlement token, then the call will succeed and this additional mint is then never recorded. This will lead to a loss of funds for the user as these funds cannot be withdrawn.

In the event that the user has no more tokens approved for the contract to transfer on their behalf, then all subsequent `long mint` calls intended to be funded via the `userAggregateBalance` will revert.

Recommendations

Modify the `if` statement brackets on line [**316**] to mitigate the issue, as shown below.

```
((commitType == CommitType.LongMint || commitType == CommitType.ShortMint) && !fromAggregateBalance)
```

Resolution

The development team has fixed this issue in PR 453.

TPP-02	Keeper Rewards Are Not Scaled Down to Settlement Token Decimals		
Asset	`implementation/KeeperRewards.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

The perpetual pools protocol utilises keepers to upkeep leveraged pools. An upkeep can be performed by anyone and is incentivized through small payments denominated in the pool's settlement token.

As settlement tokens can contain any arbitrary number of decimals fewer than 18, the `KeeperRewards.sol` contract will scale the price of the token retrieved by querying the oracle. Using this scaled amount, *{gas cost + overhead + incremental incentive}* is converted to the settlement token amount, but this is not scaled back to the settlement token's decimals. As a result, if a settlement token uses any number of decimals fewer than 18, the protocol will severely overpay keeper rewards at a net cost to the leveraged pool and its users.

Recommendations

Consider modifiying the `keeperReward()` function to return a scaled down `wadRewardValue` which uses the settlement token's decimals instead.

Resolution

The development team has fixed this issue in PR 428 by scaling keeper rewards back down to settlement token decimals.

TPP-03	Protocol Fees Can be Double Claimed		
Asset	`implementation/LeveragedPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: Medium	Likelihood: High

Description

On each update interval, a keeper will perform an upkeep which executes a price change in the asset being tracked before transferring the associated fees to the primary and secondary accounts. The `feeTransfer()` function handles how fee amounts are paid, however, the logic has been altered with the addition of the `claimPrimaryFees()` and `claimSecondaryFees()` functions.

`feeTransfer()` will transfer fees to the respective accounts and increment two storage variables, `secondaryFees` and `primaryFees`. As a result, if anyone subsequently calls `claimPrimaryFees()` or `claimSecondaryFees()`, fees will be paid out to these accounts again.

Recommendations

Consider removing the use of `safeTransfer()` in `feeTransfer()` such that fees are only transferred when `claimPrimaryFees()` or `claimSecondaryFees()` is called.

Resolution

The development team has fixed this issue by introducing the need for fees to be claimed in a separate transaction. Fees are accrued but not paid out on each upkeep. This is shown in PR 448.

TPP-04	Flip Commitments Are Not Charged Their Respective Mint And Burn Fee		
Asset	`implementation/PoolCommitter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Users are charged a fee on their tokens whenever they commit to mint or burn short or long tokens. A new commit type was introduced to allow users to flip their short tokens to long tokens and vice-versa. While it might be intended that fees should not be charged when a user flips their commit type (as the tokens remain within the protocol), it is not consistent with the implementation shown in other areas of the `PoolCommitter.sol` contract.

As a result, users can constantly rebalance their portfolio without incurring any additional cost.

Recommendations

Consider charging the relevant mint/burn fee on a flip commit. This change will likely introduce additional complexity to the protocol, so it may also be useful to keep this the same to incentivize users in two ways:

- Users can rebalance their positions at no additional cost.
- Users are incentivized to keep their money within the protocol as exiting and re-entering incurs a cost.

Resolution

The development team has mitigated this issue in PR 454 by ensuring that users who perform a flip commit are correctly charged a mint and burn fee.

TPP-05	Pool Keepers Can Re-Enter `poolUpkeep()` if Tokens With Callbacks Are Used		
Asset	`implementation/LeveragedPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: High	Impact: High	Likelihood: Medium

Description

Pool upkeeps are performed on each update interval by the protocol's keepers. The `poolUpkeep()` function will check if the pool's update interval has passed before executing a price change. However, because this function does not implement the *checks-effects-interactions* pattern, it is possible to re-enter the function and satisfy the `intervalPassed()` check again.

If we assume that the settlement token may contain a callback on transfers (typical for `ERC777` standard tokens), the secondary fee account can re-enter `poolUpkeep()` and receive fees again. This can be repeated to effectively drain the leveraged pool of its funds.

Recommendations

Ensure that `lastPriceTimestamp` is updated before fees are transferred to the primary and secondary fee addresses. Alternatively, it may be simpler to leave the implementation as is and instead modify the `feeTransfer()` function (referenced in `executePriceChange()`) to allocate funds to the fee addresses. This forces the secondary fee address to claim their fees instead, preventing them from taking control over the path of execution.

Resolution

The development team has resolved this issue by following the *checks-effects-interactions* pattern. This ensures that `lastPriceTimestamp` is updated prior to any external calls. The fix is outlined in PRs 463 and 472.

TPP-06	Stale Update Intervals Will Receive Burn Fees From Recent Update Intervals		
Asset	`implementation/PoolCommitter.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Leveraged pools function through constant upkeeps on each new update interval. An upkeep consists of *three* distinct operations:

- Price change execution (incorportating a value transfer between the long and short sides of the pool).
- Transfer of fees to the relevant addresses as part of the maintenance fee.
- Application of commitments to the leveraged pool.

When a user commits to mint tokens, the tokens and fee are instantly paid by the user, however, the relevant tokens are not minted until the next upkeep (assuming the user commits before the front-running interval). A burn commit does not follow the same method. Instead, tokens are burnt instantly, however, the tokens and fee are not calculated until the commit has been executed.

As a result, fees generated through mint commits will be instantly applied to the same side of the pool, but burn commits will only distribute fees once an upkeep has been performed. If we consider the backlog of unexecuted commitments to be greater than *one*, then the distribution of fees isn't entirely accurate. Let's consider the following example:

- We will start with *two* assumptions. The mint and burn fees are both set to *10%* and *100* settlement tokens is equal to *100* long or short tokens.
- Alice commits to mint *100* long tokens on the first update interval. Consequently, *10* long tokens are distributed to other long token holders.
- Bob commits to mint *100* long tokens on the same update interval as Alice. *10* long tokens are distributed to other long token holders.
- The update interval advances to the second interval and the first update interval is upkept by a keeper.
- Alice commits to burn her *90* long tokens on the second update interval.
- Another update interval passes such that the second update interval is unexecuted.
- Bob commits to burn his *90* long tokens on the third update interval.
- The third update interval passes without being upkept. Hence, the second and third update intervals are unexecuted and will be both executed on the next upkeep.
- Carol commits to mint *100* long tokens in which the mint fee is distributed instantly.
- A keeper peforms an upkeep on the backlogged commitments, however, because Carol has already paid her fees to the pool, Alice and Bob will still benefit be receiving a portion of fees paid by Carol.

As shown above, users could receive slightly more tokens if mint commits were made on update intervals following their burn commit and before their execution.

Recommendations

Consider utilising some sort of balance history to showcase the `longBalance` and `shortBalance` of each update interval. This will ensure the `priceHistory` mapping holds the most correct values, unimpacted by mint commits in future update intervals.

Resolution

The development team has decided not to fix this issue as per the comment below.

That case is fine. We are aware that it is (somewhat) of an exploit, but it's not a profittable enough for any rational agent to drain the pool. It also requires the pool to have >1 stale update intervals.

TPP-07	Secondary Fee Address Can Deny Pool Upkeeps by Reverting Upon Callback		
Asset	`implementation/LeveragedPool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

Pool keepers will perform upkeeps on each given update interval, ensuring that pools use the most up to date pricing for rebalance events. Upon upkeep, a fee is charged on the pool's short and long balances, used to allocate settlement tokens to the primary and secondary fee addresses.

The `executePriceChange()` function will ultimately pay fees through the `feeTransfer()` function. The settlement token is transferred according to the fee split percent. Because transfers are made during the upkeep of the pool, tokens supporting callbacks (native to `ERC777` standard tokens) will allow the secondary fee account to take control over the path of execution, allowing them to forcefully revert the upkeep.

Recommendations

This is potential opens up a Denial-of-Service vector which can be mitigated by preventing all untrusted external calls. This can be done by removing the use of `safeTransfer()` in `feeTransfer()` and instead forcing the fee recipients to claim their fees in a separate transaction.

Resolution

The development team has mitigated this security concern by removing the old fee transfer mechanism altogether and adhering to the new pattern where fees must be claimed in a separate transaction. This is outlined in PR 448.

TPP-08	Update Timestamps For `SMAOracle` And `LeveragedPool` May Diverge Due to a Backlog of Update Intervals		
Asset	`implementation/LeveragedPool.sol & implementation/SMAOracle.sol`		
Status	**Open**		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The Tracer DAO protocol is able to deal with inactive keepers by queuing commitments for execution on the next upkeep. The protocol can handle up to *16* backlogged update intervals on each upkeep. As a result, `lastPriceTimestamp` in `LeveragedPool.sol` and `lastUpdate` in `SMAOracle.sol` may diverge such that they are both no longer equal. If the amount by which these *two* timestamps diverge is greater than the front-running interval (typically set to *300* seconds), users could commit to a price outcome which is favourable to them.

A user attempting to perform such an attack only needs to delay an upkeep by *300* seconds or find a relatively inactive pool that has had its oracle frequently polled. If the attacker consistently upkeeps on each new update interval and keeps the oracle data fresh, it is likely that they can consistently arbitrage the price differential by front-running a commit to the winning side of the leveraged pool.

Recommendations

Consider restricting calls to `SMAOracle.poll()` such that they must originate from the `PoolKeeper.sol` or `KeeperRewards.sol` contracts. This will ensure the *two* update timestamps are always synced.

TPP-09	Change Interval Can be Arbitrarily Set by The Fee Controller to Brick The Leveraged Pool		
Asset	`implementation/PoolCommitter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Medium	Impact: Medium	Likelihood: Medium

Description

The `changeInterval` dictates how much the `mintingFee` should move in the positive or negative direction. This is dependent on `longTokenPrice * shortTokenPrice`. The fee controller can call `setChangeInterval()` at any time to set the `changeInterval` to any arbitrary `uint256` value. As a result, if we set this to the maximum `uint256` value, this may cause `updateMintingFee()` to revert if `PoolSwapLibrary.addBytes()` overflows and returns an invalid `bytes16` result. If `PoolSwapLibrary.compareDecimals()` utilises an invalid `bytes16` input, the comparison will revert, inhibiting the execution of `executeAllCommitments()`. This inevitably leads to a continuous a Denial-of-Service condition on any upkeep action.

Recommendations

Consider restricting `setChangeInterval()` to values less than `PoolSwapLibrary.MAX_MINTING_FEE`.

Resolution

The development team has fixed this issue in PR 456. The PR restricts `changeInterval` to `MAX_CHANGE_INTERVAL` which is effectively bounded by 100%.

TPP-10	Arithmetic Operation May Unintentionally Revert When Calculating Keeper Reward		
Asset	`implementation/LeveragePool.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The pool's upkeep is maintained by keepers, these are external actors who are incentivized to call pool functions by a reward that is then paid back to them. When calculating this reward a check is done to verify that the pool has sufficient tokens to pay out the reward in settlement tokens effectively checking `reward <= _shortBalance + _longBalance` before proceeding to calculate and pay out the reward to the keeper.

However when `reward = _shortBalance + _longBalance` the calculation in `PoolSwapLibrary.getBalanceAfterFees()` will revert. This means that keeper actions may occasionally fail when the call should succeed.

Recommendations

Amending the inequality on line [**190**] from

`amount > _shortBalance + _longBalance` to `amount >= _shortBalance + _longBalance`

will prevent this situation from occuring. Now, when the reward is exactly the same as `_shortBalance + _longBalance` the code will return `false`.

Resolution

The development team has mitigated a potential arithmetic overflow which would prevent keepers from upkeeping a pool. This is outlined in PR 454.

TPP-11	Invariant Checking Does Not Enforce a Paused `PoolCommitter` Contract		
Asset	`implementation/PoolCommitter.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The pause mechanism is to be used only when the protocol does not function as intended whether that be due to a bug or hack. When the balance invariants in `InvariantCheck.sol` do not hold true, the `PoolCommitter.sol` and `LeveragedPool.sol` contracts are paused. However, there is no enforcement of this pause mechanism in `PoolCommitter.sol`. As a result, it may still be possible to withdraw a user's claim request amount if there is no pending balance to transfer to the user. It may also be possible for users to claim via the `AutoClaim.sol` contract.

Recommendations

Consider adding the `onlyUnpaused` modifier to the `claim()` function or removing any reference of this mechanism in `PoolCommitter.sol` if this is intended behaviour.

Resolution

The development team added the `onlyUnpaused` modifier to the `claim()` function in PR 465.

TPP-12	Keepers Will be Underpaid in a Post-Merge Ethereum Environment		
Asset	`implementation/KeeperRewards.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The keeper tip amount will not prove accurate in a post-merge Ethereum environment as block times will drastically change. As a result, keepers will actually be underpaid due to the lowered per-block increase that is provided to keepers as an incentive to perform upkeeps on the perpetual pools protocol.

Recommendations

Consider allowing Tracer DAO's governance to update this parameter such that leveraged pools can be unaffected by the merge of the consensus and executions layers.

Resolution

The development team has acknowledged this issue and marked it as a *wontfix*. The `KeeperRewards.sol` can and will be updated in the future to keep up with these changes.

TPP-13	Unsafe Casting of `settlementTokenPrice`		
Asset	`implementation/KeeperRewards.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

The `payKeeper()` function performs an unsafe cast of the `settlementTokenPrice` variable. This may lead to an unlikely edge case where arithmetic underflows are not properly handled by the protocol.

Recommendations

Consider performing safe casting wherever possible. OpenZeppelin provides a useful SafeCast library which should provide the appropriate checks.

Resolution

The development team has implemented safe casting in PR 466.

TPP-14	`payKeeper()` Does Not Properly Handle Errors		
Asset	`implementation/KeeperRewards.sol`		
Status	**Resolved:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Try/catch statements are useful when we want to ensure a function handles errors without reverting. In this case, `payKeeper()` will revert if `IOracleWrapper.getPrice()` fails to correctly query the price from the oracle. As a result, `payKeeper()` may revert on this specific edge case, causing the protocol to suffer a period of downtime until the oracle functions properly again.

Recommendations

Consider using a fallback oracle or properly handle the case where the settlement token oracle does not function accordingly. Because `payKeeper()` is a sensitive function used to incentivize upkeeps, it is important that this never reverts as it impacts the overall availability of the protocol.

Resolution

The development team has acknowledged this issue and marked it as a *wontfix* as this is included in the protocol's oracle security assumptions.

TPP-15	First Committer of a New Pool Can Avoid Paying Fees		
Asset	`implementation/PoolCommitter.sol`		
Status	**Closed:** See Resolution		
Rating	Severity: Low	Impact: Low	Likelihood: Low

Description

Currently, mint commits will allocate the fee to the same side of the pool that the commit is executed on. However, because the committer is the only token holder, they will actually receive all fees generated on their own commit. This is a viable method for users to gain a position within a newly created pool without incurring any fee for this action.

Recommendations

Ensure this is understood by users if this is used as an incentive to bootstrap new leveraged pools.

Resolution

The development team has acknowledged this issue and marked it as a *wontfix*. It is intended behaviour for the mint fee to be added as *socialised* collateral to that side of the pool. As the committer consitutes the entire side of that pool, they should receive its associated fees.

TPP-16	Event Emission Always Records Zero Quantity
Asset	`implementation/AutoClaim.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

Pool users can elect to let another user initiate their minting or burning of settlement tokens on their behalf. If they do so they can incentivise this to occur by adding an `ETH` reward that goes to the caller.

In `madePaidClaimRequest()`, the situation where a caller can initiate their own action and reclaim the reward in the process is handled. Then on line [**55**], this process is recorded by the event `PaidRequestExecution`. However in recording this we use the deleted storage structure meaning that the `amount` value recorded will always be zero.

Recommendations

Replace the use of `request.reward` with the local variable copy `reward` so that the event infomation is correct.

Resolution

The development team has implemented proper event emission in PR 467.

TPP-17	Rewards Paid to Keeper Never Emitted in Events
Asset	`implementation/PoolKeeper.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The reward paid to a keeper who has called functions for the contract is calculated and returned by `KeeperRewards.payKeeper()`. However, this reward value is never recorded by the function `performUpkeepSinglePool()`. This means when the events `KeeperPaid` or `KeeperPaymentError` are emitted, they will always log the reward as zero.

Recommendations

The intended `reward` variable was introduced on line [**141**]. The output of `KeeperRewards.payKeeper()` on line [**144**] should be saved to this variable so that the events emit the accurate reward value.

Resolution

The development team has fixed the event emission in `performUpkeepSinglePool()` in PR 468.

TPP-18	There is Currently no Incentive to Continuously Monitor Pools And Perform Invariant Checking
Asset	`implementation/InvariantCheck.sol`
Status	**Resolved:** See Resolution
Rating	Informational

Description

The `InvariantCheck.sol` contract allows any user to call `checkInvariants()` when the balance invariant does not hold true. As a result, leveraged pools may be put in an unstable state for some time before a user decides to pause the contracts by calling `checkInvariants()`.

Recommendations

Consider implementing some small incentive that rewards users who monitor leveraged pools when invariants do not hold true. This will ensure affected pools are able to respond quickly to attacks on the protocol.

Resolution

The development team has recognised this as a potential concern and have mitigated this by building a social contract which rewards a user a percentage of the pool's collateral if they successfully pause the contract.

TPP-19	Inconsistent Implementation of Fees
Asset	`implementation/LeveragedPool.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

There are two types of fees in `LeveragedPool.sol`, fees charged on the entire leveraged pool, and the fee percentages used to determine how much of the fee to allocate to the primary and secondary accounts. The first `fee` type uses ABDK decimals to maximise the degree of precision, however, when splitting this fee between the primary and secondary accounts, ABDK decimals is not used and instead simple division which ultimately leads to some truncation is used instead.

Recommendations

Consider opting to use ABDK decimals for all fee calculations or ensure that it is understood that there will be a small degree of truncation when allocating fees to the respective accounts.

Resolution

The development team has acknowledged this issue and marked it as a *wontfix*.

TPP-20	`newPool()` Does Not Handle Failed Polls
Asset	`implementation/PoolKeeper.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

The `newPool()` function is called when a pool is created by the `PoolFactory.sol` contract. This initialises the first price used to perform value transfers between pool sides. In order to prevent arbitragers front-running price updates, it is essential that fresh data be used in this function.

Recommendations

Consider using a try/catch statement (with appropriate event emission) when polling the oracle. This is evident in the `performUpkeepSinglePool()` function. This will ensure the first price provided by the oracle is not stale. However, it may be intended for `IOracleWrapper.poll()` to revert without error handling. This prevents the `PoolFactory.sol` contract from deploying pools incorrectly.

Resolution

The development team has acknowledged this issue and marked it as a *wontfix*. `IOracleWrapper.poll()` is meant to revert without error handling as it prevents the deployment of incorrect pools by the factory contract.

TPP-21	Upkeeps May be Performed on Stale Oracle Data
Asset	`implementation/PoolKeeper.sol`
Status	**Closed:** See Resolution
Rating	Informational

Description

Upkeeps can be performed by any user, incentivized through payouts in the form of the pool's settlement token. Oracles are utilised when updating the price of underlying asset. A call to `IOracleWrapper.poll()` is made to ensure the data is fresh. However, due to the use of try/catch statements, poll actions may fail silently, leading to further execution of the upkeep on stale data.

Recommendations

Ensure it is understood that failed poll actions will result in the use of potentially stale oracle data when performing an upkeep.

Resolution

The development team has acknowledged this issue and marked it as a *wontfix*.

TPP-22	Miscellaneous `Tracer Perpetual Pools V2` General Comments
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

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

1. **`LeveragedPool.sol`**

   1a) Stale Code
   On line [**336**] there is commented out code that is not used, this should be removed.

   1b) Similar Variable Names
   In the function `feeTransfer()` on line [**334**] there is a local variable called `secondaryFee` and also a global variable called `secondaryFees`. It is recommended to change the name of the local variable so that it is more than one character different to the global variable to avoid confusion and risk of typos altering the output of code.

2. **`PoolCommitter.sol`**

   2a) Differing Limits for `mintingFee`
   The use of constant `MAX_MINTING_FEE` to act as a bound for `mintingFee` is inconsistent. In function `updateMintingFee()` we can set `mintingFee == PoolSwapLibrary.MAX_MINTING_FEE`. However in function `setMintingFee()` this value is not allowed as line [**881**] states `mintingFee < PoolSwapLibrary.MAX_MINTING_FEE`. The choice of `mintingFee` upper boundary should be made consistent.

   2b) Missing Event
   On line [**852**] the function `setPool()` states it emits an event on success when it does not emit any events. The missing `SettlementAndPoolChanged` event should be added or if it is not needed then the comment on line [**852**] should be removed.

   2c) Misleading Comments
   line [**628**] states `"Prevent underflow by setting mintingFee to lowest possible value (0)"` however mintingFee is an `IEEE 754` number and so supports negative numbers meaning 0 is not it's lowest possible value. The comment should be changed to something like `"Prevent underflow by setting mintingFee to lowest acceptable value (0)"`.

3. **`InvariantCheck.sol`**

   3a) Event Always Emitted
   On line [**45**] the event `InvariantsHold` is emitted regardless of the outcome. These means that if the if clause on line [**41**] has triggered then the event `InvariantsFail` will be emitted then immediately followed by the `InvariantsHold` event. This could confuse any off-chain bots or alert systems. This can be solved by adding a `break` command on line [**44**] inside of the if clause.

4. **`SMAOracle.sol`**

4a) Additional Notation Needed
On line [**72**] we have the variable `updateInterval`, this duration should have a comment outlining the expected time units it should be supplied in.

5. **PoolSwapLibrary.sol**

5a) Equation Clarity
On line [**302**] use brackets for clarity on the intended order of operation.

5b) Incorrect Comments
The comment on line [**426**] should read `"settlement tokens to return"` as no settlement tokens are being burnt.

5c) Incomplete Zero Checks
Functions `getMint()` and `getBurn()` both contain a zero price check on line [**433**] and line [**447**] respectively. However as they are checking `IEEE 754` standard numbers they should also check for negative zero.

6. **PoolFactory.sol**

6a) Constant Name
Constant `DAYS_PER_LEAP_YEAR` on line [**42**] has a misleading name as it actually records the number of seconds in a leap year.

7. **KeeperRewards.sol**

7a) Unclear Comments
The comment on line [**107**] is unclear, the `_keeperGas` variable is actually denoted in the settlement token quantity not `wei` which suggests it is an `ETH` quantity. The number format is already made clear by the `"WAD formatted"` comment. Changing the comment to `"keeper gas cost in settlement tokens."` would make it's meaning clearer.

7b) Additional Comment Required
Adding a comment to line 54 that the price returned by the oracle must always match the 18d.p. style of Chainlink ETH/Token oracles will help prevent future mistakes should a different external oracle be chosen.

8. **AutoClaim.sol**

8a) Typo
On line [**182**] `"where the all supplied"` should be `"where all the supplied"`.

9. **CalldataLogic.sol**

9a) Typo
On line [**10**] `"bite array"` should be `"byte array"`.

9b) Specify Intent
On line [**63-66**] the use of inline assembly could be improved by the clear aims of this code to help catch errors. There should be a comment documenting that due to line [**63**] the outputted `amount` is implicitly capped by `uint128.max` even though it is a `uint256`.

10. **vendors/ERC20_Cloneable.sol**

10a) Missing Zero Address Check
The function `transferOwnership()` contains a zero address check for setting `owner`, however the `initialize()` function is lacking one when `owner` is initially set.

10b) Missing Event
The function `transferOwnership()` has no event to record a transfer of ownership. This could help off-chain bots detect and monitor ownership of the pool tokens.

10c) Missing Visibility
The visibility of `_decimals` on line [**34**] is not explicitly set, this should be stated set for clarity.

Recommendations

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

Resolution

The development team has fixed the highlighted issues in PR 469.

TPP-23	Miscellaneous Gas Optimisations
Asset	`contracts/*`
Status	**Resolved:** See Resolution
Rating	Informational

Description

This section describes general observations made by the testing team in which contract callers are able to generate gas savings:

1. **Repeated Checks or Operations:**
   - In `PoolSwapLibrary.sol` the function `getMintWithBurns()` contains a zero check for price. However this zero check is then repeated by `getMint()` and `getBurn` on the same variable meaning the first check on line [**466**] can be safely removed.
   - On line [**141**] of `KeeperRewards.sol` we make an unnecessary casting of `_settlementTokenPrice` to a `uint256` when it is already a `uint256`.
   - On line [**93**] of `SMAOracle.sol` the require statement is unnecessary as in solidity 0.8.7 line [**95**] will revert in the event that `inputOracleDecimals > decimals`.
   - The incrementation `i++` on line [**185**] of `SMAOracle.sol` can be placed inside an unchecked block at the end of the loop to save gas as it's value is bounded by `periodCount` and so cannot overflow.
   - On line [**756**] of `implementation/PoolCommitter.sol` we delete `unAggregatedCommitments[user]` however this is overwritten on the next line. Due to how Solidity handles overwriting with dynamic length arrays the `delete` operation is unneeded.
   - The `KeeperRewards.payKeeper()` function will poll the settlement token oracle before retrieving its price. However, this can be optimised by utilising the result of the `IOracleWrapper.poll()` call which will return the SMA if `SMAOracle.sol` is used. If it is expected that both `SMAOracle.sol` and `ChainlinkOracleWrapper.sol` may function as the oracle endpoint, then it may be useful to remove the `SMAOracle._calculateSMA()` call in `SMAOracle.poll()` and avoid returning the unused `int256`.
2. **Unused Memory Copy of Storage Variable:** Several times values are written to local variables which are then only used to write the same value to a storage variable. A small gas saving can be had by directly writing to storage in these instances.
   - On line [**194**] of `LeveragedPool.sol` we introduce local variables `shortBalanceAfterRewards` and `longBalanceAfterRewards` however these are only used to write to the storage variables `shortBalance` and `longBalance` and so can be removed.
   - On line [**66**] of `AutoClaim.sol` we store the outcome of `poolCommitter.getAppropriateUpdateIntervalId()` in a local variable which is then just written to storage on line [**67**], the local variable is then never read again and so the outcome could just be directly written to storage instead.
3. **Event Emission Reordering:** The functions `updateFeeAddress()`, `updateSecondaryFeeAddress()`, `setKeeper()` and `claimGovernance()` of `LeveragedPool.sol` and `claimGovernance()` of `PoolFactory.sol` could all move the event emission prior to overwriting of their related storage variable to avoid also needing a local variable. While this breaks the `"Checks, Effects and Interactions"` coding pattern advice; in these instances not following the CEI pattern poses no additional risk.

4. **Conditional Minimal Value Setting:**
   In a specific case, we aim to set a variable `x` to `Min(x,y)` for two `uint256` values `x` and `y`, this can be done more efficiently using a bitwise `AND` operation as then larger bits are discarded.
   - In `PoolCommitter.sol` the conditional operator on line [**708**] can be replaced with `update._maxIterations = uint8(unAggregatedLength) & MAX_ITERATIONS;` This has exactly the same behaviour but can save up to 16900 gas for a 46% reduction.
5. **Storage Accesses Can Be Replaced By Memory Operations:**
   In `SMAOracle.sol` on line [**185**] we use a global variable `periodCount` twice in a loop, `periodCount` should be stored in a local variable prior to beginning the loop and this should be accessed instead to avoid unnecessary `SLOAD` operations.
6. **Variables Can Be Made Immutable For Gas Savings:**
   Both the `string adminRoleDescription` and `bytes32 adminRoleDescriptionHash` variables can utilise the `immutable` keyword to generate small gas savings.
7. **Stale Code:**
   - On line [**13**] of `SMAOracle.sol` we use `PRBMathSD58x18` however this library's functions are never actually used and so can be removed.
   - In `CalldataLogic.sol` The constants `SLOT_LENGTH`, `FUNCTION_SIGNATURE_LENGTH`, `SINGLE_ARRAY_OFFSET` and `DOUBLE_ARRAY_OFFSET` are never used in this library nor imported by any other contracts and so should be removed.
   - The function `transferOwnership()` of `vendors/ERC20_Cloneable.sol` is only callable by `PoolFactory` but is never called. Either this function should be removed or it's use implemented somewhere in `PoolFactory`.
8. **Functions Can Be Made External For Gas Savings:**
   In `PoolCommitter.sol` the functions `getMintingFee()` and `getBurningFee()` can be marked as external functions as neither are called internally by any other functions of `PoolCommitter.sol`. This will generate gas savings on contract deployment and on each function call.
9. **Function Rewrites:**
   Several functions can have sections altered to reduce gas costs without altering their outcomes.
   - In `SMAOracle.sol` we calculate the sum of the last `k` prices on each call. This requires up to 24 (The maximum value of `numPeriods`) variables be loaded from storage each time. Instead you could store the sum and simply remove the oldest price and add on the newest price to update the sum on each call. Then every call after the first full average calculation would then require only 3 variables be loaded from storage and would likewise reduce the number of addition operations needed.
   - In `SMAOracle.sol` if we store the variables `_numPeriods`, `numPeriods` and `periodCount` as signed integers, then we don't need to cast `k` from `uint256` to `int256` on line [**190**]. `_numPeriods`, `numPeriods` have their given value capped by `MAX_PERIOD` and if `periodCount` represents seconds then even as a `int256` it can store values expressing over 1.83×10^{69} years.
   - In `implementation/LeveragedPool.sol` on line [**274**] we pack the inputted price change data into a structure before then handing it to `PoolSwapLibrary.calculatePriceChange()` on line [**287**]. Once there it is then unpacked and saved to local variables. This process isn't useful and just adds to the gas costs.
10. **Temporary Storage Variables Can Be Deleted:**
    `PoolFactory.claimGovernance()` and `LeveragedPool.claimGovernance()` can delete the storage variable `provisionalGovernance` for an additional gas refund. However, it might make sense to keep the slot warm as it reduces any subsequent call to `transferGovernance()`.
11. **Inline Code Optimisations:**
    line [**111**] and line [**113**] of `PoolKeeper.performUpkeepSinglePool()` can be combined such that we do not load an external call into memory and then read from memory. Instead, the external call can be read directly.

Recommendations

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

Resolution

The development team has fixed the highlighted issues in PR 470.

Appendix A Test Suite

A non-exhaustive list of tests were constructed using the `foundry` framework to aid this security review and are given along with this document.

```
Running 14 tests for contracts/test/FuzzPoolSwapLibrary.t.sol:PoolSwapLibraryTest
[PASS] testConvertDecimalToUInt(bytes16) (runs: 1, μ: 8903, ~: 8903)
[PASS] testConvertDecimalToUIntSaneInputs(uint256) (runs: 1, μ: 4872, ~: 4872)
[PASS] testFuzzAddSubMultiplyCompareBytes(uint256,uint256) (runs: 1, μ: 13070, ~: 13070)
[PASS] testFuzzConvertUInttoDecimal(uint256) (runs: 1, μ: 7043, ~: 7043)
[PASS] testFuzzDivUInt(uint256,uint256) (runs: 1, μ: 6960, ~: 6960)
[PASS] testFuzzGetBalancesAfterFees(uint256,uint256,uint256) (runs: 1, μ: 5924, ~: 5924)
[PASS] testFuzzGetBalancesAfterFeesEqualsOnly(uint256,uint256) (runs: 1, μ: 5398, ~: 5398)
[PASS] testFuzzGetBalancesAfterFeesGreaterOnly(uint256,uint256,uint256) (runs: 1, μ: 5861, ~: 5861)
[PASS] testFuzzGetLossAmount(uint256,uint256,uint256) (runs: 1, μ: 16714, ~: 16714)
[PASS] testFuzzGetLossMultiplier(uint8,uint8,uint8) (runs: 1, μ: 10348, ~: 10348)
[FAIL. Reason: Call did not revert as expected] testGetBurn() (gas: 9736)
[PASS] testGetMint() (gas: 9798)
[FAIL. Reason: Call did not revert as expected] testGetMintWithBurns() (gas: 15910)
[PASS] testGetRatio(uint256,uint256) (runs: 1, μ: 5378, ~: 5378)
Test result: FAILED. 12 passed; 2 failed; finished in 3.66ms

Running 5 tests for contracts/test/FuzzLeveragedPool.t.sol:LeveragedPoolTest
[PASS] testFuzzCommitSequence(uint256[10],uint16[10],bool[10]) (runs: 1, μ: 2083394, ~: 2083394)
[PASS] testFuzzValueBleed(uint256[100],bool[100]) (runs: 1, μ: 18056333, ~: 18056333)
[PASS] testPoolCommits() (gas: 2522150)
[PASS] testPoolRebalance(int256) (runs: 1, μ: 1097687, ~: 1097687)
[PASS] testPoolUpkeep() (gas: 282448)
Test result: ok. 5 passed; 0 failed; finished in 8.42s

Failed tests:
[FAIL. Reason: Call did not revert as expected] testGetBurn() (gas: 9736)
[FAIL. Reason: Call did not revert as expected] testGetMintWithBurns() (gas: 15910)
```

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.

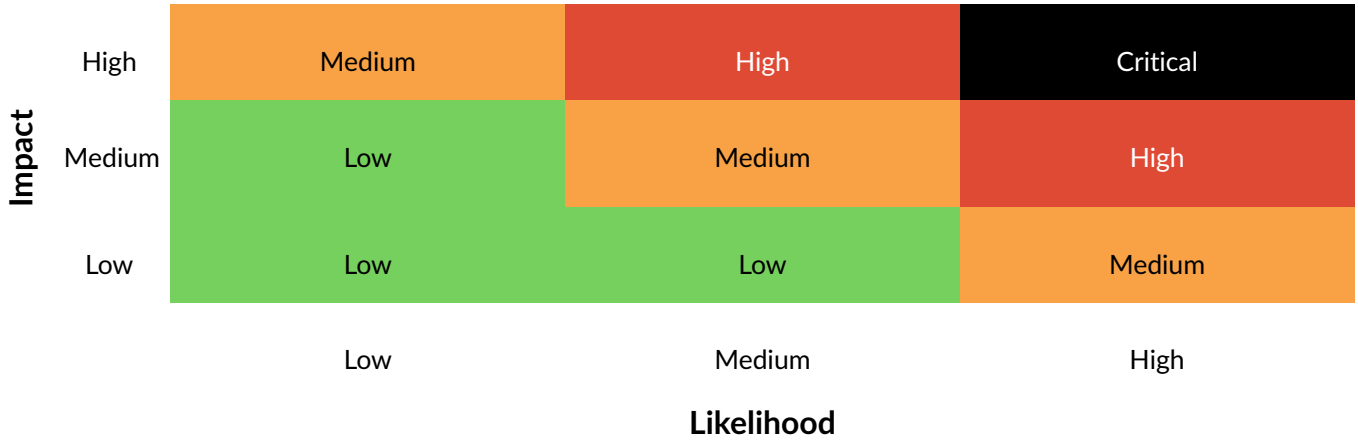

Table 1: Severity Matrix - How the severity of a vulnerability is given based on the *impact* and the *likelihood* of a vulnerability.