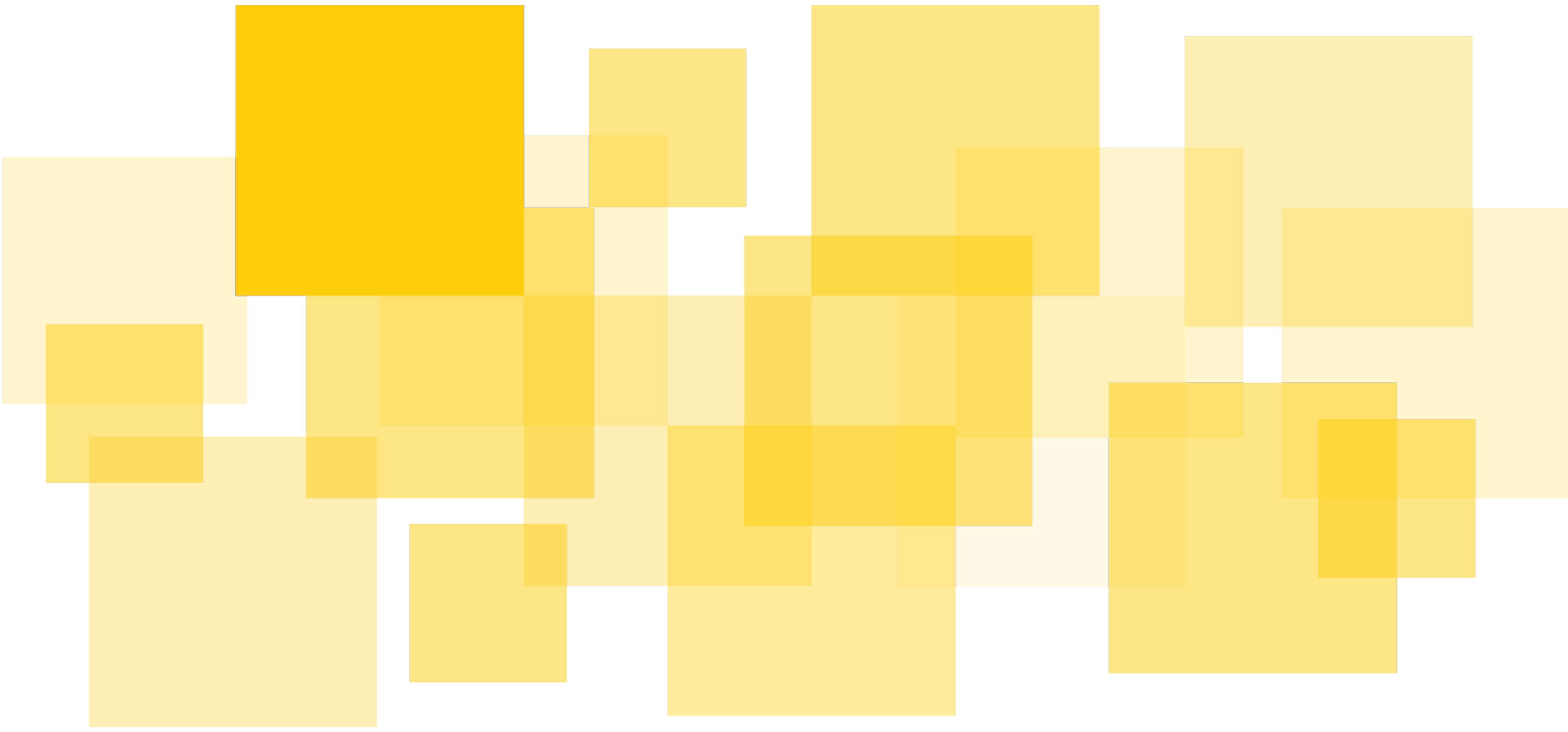
Security Audit Report

Tracer Perpetual Pools V2 Contract

Delivered: December 17th, 2021



Prepared for Tracer by

runtime verification

Summary

Runtime Verification, Inc. conducted a security audit on the Tracer Perpetual Pools V2 and vesting smart contracts. The audit was conducted by Raoul Schaffranek, Daejun Park, and David Kretzmer from October 25, 2021, to December 17, 2021.

Several issues have been identified as follows:

- Implementation flaws: A06, B12, B13
- Potential security vulnerabilities: A01, A02, A03, A04, A05, A07, A08

A number of additional suggestions have also been made, including:

- Input validations: B09, B10
- Best practices: B01, B02, B03, B11
- Gas optimization: B14
- Code readability: B05, B06, B07, B08, B15, B16, B17, B18, B19

All the critical security issues have been addressed by the client. Details can be found in the Findings section as well as the Informative Findings section.

The code is generally well written and thoughtfully designed, following best practices. One point of concern is the usage of a third-party floating-point library, which we discuss in the Floating-Point Arithmetic section.

Scope

The targets of the audit are the smart contract source files at git-commit-id 846bbf62652d7c83aee1cf3766275c4d08b00c8a (Perpetual Pools Contract) and a02cbd5e73e629b5d80c0af0202a3ee6f18d0216 (Vesting Contract).

The audit focused on the following core contracts and interfaces:

- Vesting.sol
- ChainlinkOracleWrapper.sol
- PoolFactory.sol
- PoolToken.sol
- LeveragedPool.sol
- PoolKeeper.sol
- PriceObserver.sol
- PoolCommitter.sol
- PoolSwapLibrary.sol

- SMAOracle.sol
- IERC20DecimalsWrapper.sol
- IHistoricalOracleWrapper.sol
- ILeveragedPool.sol
- IOracleWrapper.sol
- IPoolCommitter.sol
- IPoolFactory.sol
- IPoolKeeper.sol
- IPoolToken.sol
- IPriceObserver.sol

The audit is limited in scope within the boundary of the Solidity contract only. Off-chain and client-side portions of the codebase, as well as deployment and upgrade scripts are *not* in the scope of this engagement but are assumed to be correct (see below).

Assumptions

The audit is based on the following assumptions and trust model.

- External token contracts conform to the ERC20 standard, especially that they *must* revert in failures. They do *not* allow any callbacks (e.g., ERC721 or ERC777) or any external contract calls. They do *not* implicitly update the balance of accounts (e.g., rebasing tokens or fees on transfers) other than explicit events of transfers, mints, or burns.
- The contracts are operated by multiple parties, and the secure operation of the contracts requires all the participating agents to behave correctly, honestly, and diligently.
- The contracts will be deployed and integrated correctly.

Methodology

Although the manual code review cannot guarantee to find all possible security vulnerabilities as mentioned in Disclaimer, we have followed the following approaches to make our audit as thorough as possible. First, we rigorously reasoned about the business logic of the contract, validating security-critical properties to ensure the absence of loopholes in the business logic and/or inconsistency between the logic and the implementation. To this end, we developed a high-level model of the central logical parts. Then, we carefully checked if the code is vulnerable to known security issues and attack vectors. Finally, we symbolically executed part of the compiled bytecode to systematically search for unexpected, possibly exploitable, behaviors at the bytecode level due to EVM quirks or Solidity compiler bugs.

The high-level model can be found online at:
https://gist.github.com/daejunpark/545d43f02661f3f49fb500aa3cc2be5a

Floating-Point Arithmetic

Solidity offers no native data type for floating-point arithmetic, consequently, developers are forced to fall back to userland implementations. This increases the burden of library implementors and users if they depend on functions that cannot be carried out by fixed-point or integer arithmetic as is the case for the pool contracts that are the subject of this audit. The contracts depend on a library called ABDKMathQuad. The library code is outside the scope of this engagement, and it raised some general concerns:

1. The library implementation is inherently complex (for good reasons), and hasn't been thoroughly reviewed, tested, intensively used, or formally verified, to the best of our knowledge.
2. The floating-point arithmetic is much more complex and unpredictable than fixed-point or integer arithmetic, and especially the rounding error analysis is much harder. It is nontrivial to use correctly and safely, avoiding all the pitfalls.
3. Converting between integers and floating-point values introduces another potential source of subtle bugs, which are difficult to spot and hard to track down.
4. At the current gas price, gas-efficient code becomes a major challenge. A library implementing floating-point arithmetic at the userland level cannot archive the same gas efficiency as a native solution on the bytecode level.

For these reasons, we want to hand out the following general recommendations:

1. If possible choose libraries, which have been tested, audited, intensively used, and formally verified.
2. Don't use the floating-point library when native arithmetic can do the job. We have identified two function calls where neither integer nor fixed point arithmetic can be used, `ABDKMathQuad.pow_2()` and `ABDKMathQuad.log_2()`. However, please note that the list is not necessarily exhaustive. Many other uses of the library could in principle be performed on native arithmetic.

Disclaimer

This report does not constitute legal or investment advice. The preparers of this report present it as an informational exercise documenting the due diligence involved in the secure development of the target contract only and make no material claims or guarantees concerning the contract's operation post-deployment. The preparers of this report assume no liability for any and all potential consequences of the deployment or use of this contract.

Smart contracts are still a nascent software arena, and their deployment and public offering carries substantial risk. This report makes no claims that its analysis is fully comprehensive and recommends always seeking multiple opinions and audits.

This report is also not comprehensive in scope, excluding a number of components critical to the correct operation of this system.

The possibility of human error in the manual review process is very real, and we recommend seeking multiple independent opinions on any claims which impact a large quantity of funds.

Findings

A01: Unsafe transfer of ERC20 tokens

[Severity: High | Difficulty: - | Category: Security]

Disclaimer

The vesting and pool contracts can, in principle, interact with arbitrary ERC20 tokens. Unfortunately, not all ERC20 implementations conform to the standard specification. Interaction with these malfunctioning tokens is *not* in the scope of the audit. Please refer to the assumptions section for further details. As a consequence, this finding should be considered as the best effort to point to an issue that came to our attention regarding the USDT token.

Description

The USDT token does not conform to the ERC20 standard. In particular, the transfer functions do not return boolean values. Consequently, all transfers of USDT tokens initiated by the contract will eventually revert because the surrounding success checks will fail.

Recommendation

Do not expect that transfer calls made to arbitrary ERC20 tokens will return boolean values. Consider using a library, e.g., OpenZeppelin's SafeTransferERC20, to work around the issue.

Status

Acknowledged by the client.

A02: `SMAOracle.update()` never updates `lastUpdate` timestamp

[Severity: High | Difficulty: Low | Category: Security]

Price updates should be executed exactly once per `updateInterval`. Therefore, `SMAOracle` maintains a storage variable `lastUpdate` to keep track of the last successfully completed price update. The `poll()` function will revert if it is called before the next update interval has passed. However, the `lastUpdate` variable is never updated. Consequently, all calls to `poll()` will always succeed after the first update interval. A malicious user can exploit this vulnerability for price manipulations.

Recommendation

The `update()` function should set the value of the `lastUpdate` variable to the current `block.timestamp`.

Status

Acknowledged by the client.

A03: `PoolCommitter.claim()` Potential reentrancy vulnerability

[Severity: High | Difficulty: High | Category: Security]

The `claim()` function allows a user to withdraw his locked-in settlement-, short- and long-tokens. The settlement token can be an arbitrary ERC20-token. Hence, an external call is made to transfer the token from the contract to the end-user. Without making any further assumptions about the settlement-token implementation, it is possible that the transfer function can call back into the `PoolCommitter.claim()` function. Now, the internal balance of the user's tokens is updated only after the external token transfer. Consequently, if a user manages to re-enter into the `claim()`-function, he will be able to drain all tokens from the contract.

Scenario

1. Assume Alice has a balance of 100 settlement tokens. The total locked-in supply of settlement tokens is 1000.
2. Alice calls `claim()` to withdraw her funds:
    a. An external call to the transfer function of the settlement token is made.
    b. If the pool has less than 100 tokens, go to step 3.
    c. Otherwise, the target contract transfers 100 tokens from the pool contract to Alice.
    d. The target contract calls back into `PoolCommitter.claim()`
    e. Goto step 2.a.
3. Alice now owns 1000 settlement tokens. The pool contract has been drained.

Recommendation

We recommend following the checks-effects-interactions pattern and updating the internal balance before making external calls.

Status

Acknowledged by the client.

A04: `PoolCommitter.claim()` griefing vulnerability

[Severity: Low | Difficulty: Low | Category: Security]

The PoolCommitter offers two features that can be abused in conjunction by an attacker to cause grief on other users:

1. The PoolContract allows every user to withdraw balances on other users' behalf. For example, if Bob withdraws on Alice's behalf, Alice's balance will be transferred to her.
2. When minting long or short tokens, users can opt to lock in the required collateral from their balance.

Consequently, Bob can withdraw on Alice's behalf to hinder her from minting if she intends to lock in the collateral from her balance.

Scenario

1. Assume Alice has an aggregated balance of 100 settlement tokens.
2. Alice commits to minting long (or short) tokens. The collateral for the mint should be taken from her aggregated balance.
3. Bob sees Alice's transaction on the mempool.
4. Bob makes a call to claim(Alice) with higher transaction fees than Alice's transaction.
5. If Bob's transaction is included before Alice's, Alice's minting operation will be unsuccessful because she provides zero collateral.

Notice that Bob has no economic incentive in this scenario. Moreover, Alice can still mint tokens by transferring fresh collateral to the contract instead of taking it from her aggregated balance.

Recommendation

Consider implementing an approval scheme so that only users who have been granted explicit approval can withdraw on behalf of the approver.

Status

Acknowledged by the client.

A05: `PoolCommitter.executeCommitments()` - Potential DoS due to unbounded loop

[Severity: High | Difficulty: High | Category: Security]

The `executeGivenCommitments()` function of the `PoolCommitter` contract contains an unbounded loop that iterates over the pending commitments. If the `lastPriceTimestamp` becomes very old, this could lead to a condition where the loop can no longer be executed because its gas consumption exceeds the block gas limit. Consequently, the contract becomes permanently inoperable.

Recommendation

Our suggestion is to add another function that will only execute the first iteration of the loop. This way, it is always possible to recover from the condition as mentioned above by executing iteration steps one after another in isolated transactions. An alternative solution would be to add a limit parameter that bounds the number of loop iterations.

Status

Acknowledged by the client.

A06: `PoolCommitter.claim(address user)` updates the aggregate balance of the wrong user

[Severity: Low | Difficulty: Low | Category: Implementation flaws]

The `PoolCommitter` contract keeps track of users' balances. This balance is updated automatically whenever a user commits or claims his internal balance. Additionally, it can be updated manually by the user. However, when a user claims his balance, the update is performed on the message sender's balance instead of the target user's. As a consequence, a user might receive less than his total balance. Notice that a user can still claim his outstanding balance by updating his internal balance manually before claiming again.

Recommendation

Modify the `claim()` function to automatically update the target user's balance instead of the message senders.

Status

Acknowledged by the client.

A07: PoolCommitter - Bypass front-running interval

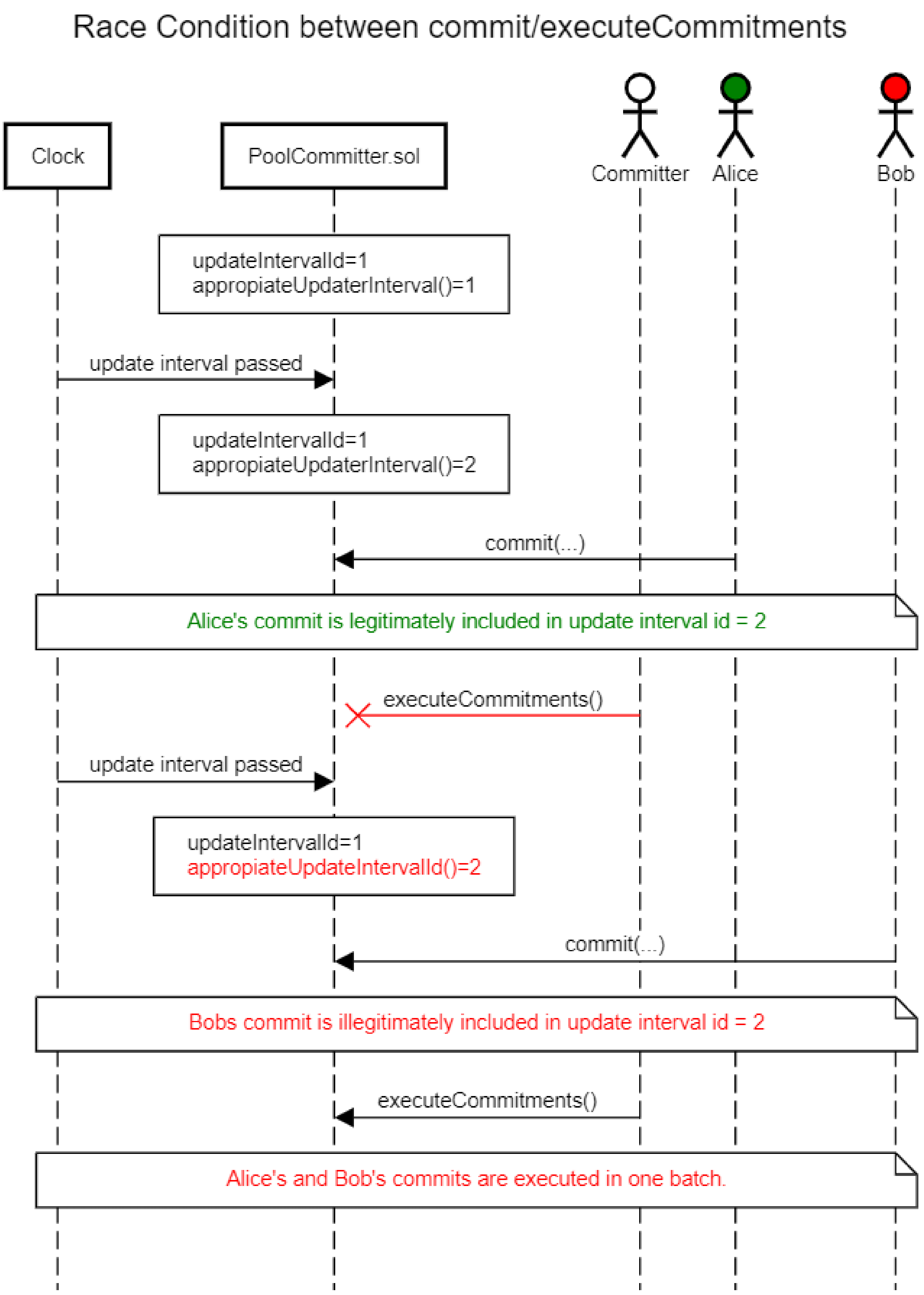
[Severity: High | Difficulty: High | Category: Security]

The PoolContract defines a front-running interval to prevent users from observing the price oracle and committing to a predictable price. However, if the front-running interval is lower than the update interval and the pool keeper fails to execute pending commitments timely, it is possible to bypass the front-running interval.

Scenario

Please refer to the diagram below to understand the following scenario.

1. Assume `poolCommitter.updateIntervalId = 1`
2. An update interval passed.
3. Alice makes a call to `commit()`. Her commitment is legitimately assigned to `updateIntervalId = 2`.
4. The committer fails to execute the commitment on time.
5. Another update interval passes.
6. Bob makes a call to `commit()`. His commitment is illegitimately assigned to `updateIntervalId = 2`. This is because `appropiateUpdateIntervalId()` does not take into account how many update intervals have passed since the last successful commitment execution. At this point, Bob can predict the next price change by observing the price oracle.
7. The committer calls `executeCommitments()` successfully. Notice, this event can also be initiated by Bob, if calls `poolKeeper.performUpkeepSinglePool()`.
8. Alice's and Bob's commitments are executed as part of the same interval and consequently on the same pricing data.


Race Condition between commit/executeCommitments
Clock
PoolCommitter.sol
Committer
Alice
Bob
updateIntervalId=1
appropiateUpdaterInterval()=1
update interval passed
updateIntervalId=1
appropiateUpdaterInterval()=2
commit(...)
Alice's commit is legitimately included in update interval id = 2
executeCommitments()
update interval passed
updateIntervalId=1
appropiateUpdateIntervalId()=2
commit(...)
Bobs commit is illegitimately included in update interval id = 2
executeCommitments()
Alice's and Bob's commits are executed in one batch.

Recommendation

Adjust the logic of `appropiateUpdateIntervalId()` to account for the number of update intervals that have passed since the last successful commit execution.

Status

Acknowledged by the client.

A08: PoolCommitter - incorrect calculation of shadow balances

[Severity: High | Difficulty: High | Category: Security]

If the front-running interval is higher than the update interval, and the pool keeper fails to execute pending commits on time, the shadow balance is calculated incorrectly and pending commits might be executed at a wrong price.

Scenario

1. Suppose that `updateInterval = 10`, and `frontRunningInterval = 20`.
2. Suppose that the current `updateIntervalId = 100`, and `lastPriceTimestamp = 990` at the moment. Also, suppose that the `longTotalSupply` (i.e., `IERC20(tokens[0]).totalSupply()`) is `30`, and the current `pool.longBalance() = 300`. That is, the current long price is `10 (= 300 / 30)`.
3. Alice calls `commit(LongBurn(5))` when `block.timestamp = 995`. At this point, `appropriateUpdateIntervalId = 102`, and Alice's `LongBurn(5)` commit is recorded in `totalPoolCommitments[102]`. Now, the `longTotalSupply = 25`.
4. Bob calls `commit(LongBurn(15))` when `block.timestamp = 1005`. At this point, `appropriateUpdateIntervalId = 103`, and Bob's `LongBurn(15)` commit is recorded in `totalPoolCommitments[103]`. Now, the `longTotalSupply = 10`.
5. Suppose that there are no other commits made until `block.timestamp = 1035`. Also, let's assume that there is no price change in the meantime for simplicity. Then, `executeCommitments()` is called, which will eventually call `executeGivenCommitments(totalPoolCommitments[102])` and `executeGivenCommitments(totalPoolCommitments[103])`.
6. In `executeGivenCommitments(totalPoolCommitments[102])`, `longTotalSupplyBefore = 10`, but `totalLongBurn` is only `5`. Thus, the long `priceHistory[102]` is `20 (= 300 / (10+5))`, which is incorrect, where the correct price is `10`. Now, the long balance becomes `200 (= 300 - 5 * 20)`, which is incorrect, where the correct value is `250 (= 300 - 5 * 10)`.
7. Also, in `executeGivenCommitments(totalPoolCommitments[103])`, `longTotalSupplyBefore = 10`, but `totalLongBurn` is only `15`. Thus, the long `priceHistory[103]` is `8 (= 200 / (10+15))`, which is incorrect, where the correct price is `10`. Now, the long balance becomes `80 (= 200 - 15 * 8)`, which is incorrect, where the correct value is `100 (= 300 - 5 * 10 - 15 * 10)`.
8. So, Alice gets `100` which is more than it should (`50`), while Bob gets `120` which is less than it should (`150`). And the pool loses.
9. This behavior can be exploited if Alice is able to control the network for a short period of time or to bribe all the potential keepers to not upkeep for a while. So, the severity is HIGH, but the difficulty is also HIGH.

Recommendation

For the short term, adjust the logic of the shadow balance calculation to handle updates gracefully when no commit execution happened during an update interval.

For the long term, consider simplifying the accounting logic, e.g., by *not* burning tokens at the time of commitment, but temporarily transferring them to an escrow account, and later burning them when the commitment is actually executed.

Status

Acknowledged by the client.

A09: `SMAOracle.getPrice()` returns the wrong price when not enough prices are observed

[Severity: High | Difficulty: Low | Category: Security]

In SMAOracle.SMA(), `xs.length` always returns `24`, even if xs is partially filled. Consequently, `getPrice()` will return an incorrect value if the number of observed prices is less than `24` (i.e., up to the 24 time units passed since the SMAOracle deployment, or the last `PriceObserver.clear()` call.)

Scenario

1. Suppose `SMAOracle` is just deployed with periods being set to `8`.
2. Suppose we have eight price observations, where all are equal to `10`.
3. Now, `getPrice()` will return `0`, which is incorrect, where the correct answer is `10`.
4. Note that `getPrice()` will work correctly, only after we've seen at least `24` price observations since the `SMAOracle` deployment or the last `PriceObserver.clear()` call.

Recommendation

Add an additional parameter, say `length`, in the SMA() function, which is to be fed by `priceObserver.length()`.

For example: `SMA(priceObserver.getAll(), priceObserver.length(), periods);`

Status

Waiting for client’s feedback.

Informative Findings

B01: Vesting - Recycling of canceled schedules

[Severity: Low | Difficulty: - | Category: Best Practice]

The owner of the vesting contract can cancel vesting schedules if they’re not flagged as `isFixed`. To cancel a schedule the owner calls the `rug()` method. Currently, the implementation of `rug()` sets `schedule.totalAmount = 0`, and keeps the schedule in storage.

Recommendation

We recommend deleting the schedule from the storage to minimize the attack surface of potential recycling attacks.

Status

Acknowledged by the client.

B02: Vesting - Sanity check of scheduleId

[Severity: Low | Difficulty: - | Category: Best Practice]

The vesting contract maintains mappings from identifiers to schedules for each user. When such a mapping is accessed, there are always two cases to consider: 1) The identifier points to an existent schedule 2) The identifier points to a non-existent schedule. If the second case is not handled explicitly it establishes a potential source of undefined behavior.

Recommendation

We recommend adding sanity checks for schedule identifiers when accessing the internal mappings and explicitly handling both cases (existent/non-existent schedule) via some branching structure (require/if). Since there is no canonical way to test structs for existence in Solidity, we recommend choosing one (or a combination) of the following defensive programming techniques:

1) Add an `existent` flag to schedule structs. The flag is set during `vest()`. The `rug()` method should unset the flag or delete the schedule from storage. Examples:

   For `claim`: `require(schedules[msg.sender][scheduleNumber].existent)`
   For `rug: require(schedules[account][scheduleId].existent)`

2) Derive the existence of a schedule from its total amount. Examples:

   For `claim`: `require(schedules[msg.sender][scheduleNumber].totalAmount > 0)`
   For `rug: require(schedules[account][scheduleId].totalAmount > 0)`

3) Compare to `numberOfSchedules` to get a conservative approximation of schedule existence. Examples:

   For `claim: require(schuduleNumber < numberOfSchedules[msg.sender])`
   For `rug: require(scheduleId < numberOfSchedules[account])`

Status

Acknowledged by the client.

B03: Vesting - Bounds checking in `calcDistribution()`

[Severity: Low | Difficulty: - | Category: Best Practice]

The vesting contract uses the function `calcDistribution()` to compute the number of vested tokens. The function currently does not take into account the `cliffTime` and `endTime` boundaries of a schedule. Currently, the function is only used by the `claim()` method, which performs the bounds checking correctly. However, if the function will be re-used in a future revision of the contract, the bounds-checking must be re-implemented.

Recommendation

Move the bounds checking inside the `calcDistribution` function. In particular, the function should return `0`, if `currentTime <= cliffTime`, and it should return `amount` if `endTime <= currentTime`.

Status

Acknowledged by the client.

B04: Vesting - Possible front-running of `rug()`

[Severity: Low | Difficulty: Low | Category: Security]

Users can observe the mempool to see if they're going to be rugged. If they see a corresponding transaction, they can claim their outstanding vested tokens before they get rugged.

Scenario

1. The owner creates a vesting schedule for Alice.
2. The owner calls `rug()` at some point in time t with `cliffTime <= t`.
3. Alice sees the transaction on the mempool and calls claim with higher gas fees.
4. If Alice's tx is included before the owner's, Alice gets transferred her vested tokens. If `endTime <= t,` she will get the complete outstanding amount.

Recommendation

Document the above scenario clearly.

Status

Acknowledged by the client. Tracer indicated that this is acceptable behavior.

B05: SMAOracle - never uses `prb-math` library

[Severity: Low | Difficulty: - | Category: Code Readability]

The `SMAOracle.sol` file imports a third-party library called `prb-math/contracts/PRBMathSD59x18.sol`, which is never used.

Recommendation

Remove the unused import.

Status

Acknowledged by the client.

B06: `SMAOracle.update()` directly refers to `PriceObserver` implementation rather than the interface

[Severity: Low | Difficulty: - | Category: Code Readability]

The `SMAOracle` contract holds a storage variable `priceObserver` that points to the implementation address of an `IPriceObserver` interface. The `update()` function casts the address to a `PriceObserver` contract, which is a specific implementation of the interface mentioned above. However, it is not given that the target address is indeed an instance of the `PriceObserver` contract. In theory, it could be any other implementation of the (or even not an interface implementation at all). Since all calls to the `priceObserver` are external calls, we cannot assume that the code will be loaded from a `PriceObserver` instance. This is a potential source of confusion for people trying to follow the code.

Recommendation

We recommend avoiding casting the `priceObserver` storage variable directly to some concrete implementation. Instead, cast it into the `IPriceObserver` interface. This way, it becomes clear that no actual code is linked to the external calls.

Status

Acknowledged by the client.

B07: `SMAOracle.toWad()` is never used

[Severity: Low | Difficulty: -| Category: Code Readability]

The `SMAOracle` contract defines a function `toWad()`, which is the inverse operation of `fromWad()`. However, the function is declared private, is never used internally, and is also not part of the interface specification of `IOracleWrapper`. Consequently, the function contains unreachable code.

Recommendation

If there is a valid use case for `toWad()`, consider declaring it external. Otherwise, consider removing it.

Status

Acknowledged by the client.

B08: `SMAOracle.price` is never used

[Severity: Mid | Difficulty: - | Category: Code Readability]

The `SMAOracle` contract defines a storage variable `price` that is never used. It might be confusing to readers of the code to see this unused storage variable, especially since the Solidity compiler will automatically generate a getter function, called `price()`, to read the current value of the `price` variable. The actual current price is retrieved from the `getPrice()` function. The latter function will not have the intended semantics, as it will always return `0`.

Recommendation

Remove the unused storage variable.

Status

Acknowledged by the client.

B09: `SMAOracle.constructor()` missing zero-check for `_deployer` argument

[Severity: Low | Difficulty: - | Category: Input Validation]

The `SMAOracle` contract defines a public storage variable `deployer`. The variable holds an address, which will be the only address with the privilege to create a pool (via the `PoolFactory` contract) pointing to this specific oracle instance. Consequently, the oracle cannot be used to create pools if the deployer address is zero.

Recommendation

Add a zero-check for `_deployer` in the constructor of `SMAOracle` to prevent the deployment of oracles that cannot be used for pool deployment.

Status

Acknowledged by the client.

B10: `ChainlinkOracleWrapper.constructor()` missing zero-check for `_deployer` argument

[Severity: Low | Difficulty: - | Category: Input Validation]

The `ChainlinkOracleWrapper` contract defines a public storage variable `deployer`. The variable holds an address, which will be the only address with the privilege to create a pool (via the `PoolFactory` contract) pointing to this specific oracle instance. Consequently, the oracle cannot be used to create pools if the deployer address is zero.

Recommendation

Add a zero-check for `_deployer` in the constructor of `ChainlinkOracleWrapper` to prevent the deployment of oracles that cannot be used for pool deployment.

Status

Acknowledged by the client.

B11: `PoolCommitter.executeCommitments()` Missing check for `updateInterval`

[Severity: Mid | Difficulty: Difficult | Category: Best Practice]

The `executeCommitments()` function always executes the first `totalCommitment` without checking if `lastPriceTimestamp + updateInterval` has passed. Consequently, if this function is called multiple times, it can be used to execute all pending commitments. In particular, this includes those commitments that were scheduled in the future for which the `frontRunningInterval` has passed.

Discussion

The function is only callable from the pool address, and the only call is guarded by a check, which ensures that the respective `updateInterval` has passed. Consequently, there is no obvious way to exploit this behavior.

Recommendation

We recommend adding a sanity check to the `executeCommitments()` function as a defense programming technique, anticipating future uses of the function.

Status

Acknowledged by the client.

B12: PoolCommitter - Recycling of personal commit histories

[Severity: Low | Difficulty: - | Category: Implementation flaws]

The `updateAggregateBalance` of the `PoolCommitter` contract is used to compute the internal balance of settlement-, short- and long-tokens which the user owns. The information is derived from the users' commit history. After successful computation, the involved commitments should be removed from the history to decrease the attack surface of potential recycling attacks. The removal operation is present in the code, but it is called with the wrong parameters.

Recommendation

Change the line of code to delete commits from users' histories from `delete userCommitments[user][i]` to `delete userCommitments[user][id]`.

Status

Acknowledged by the client.

B13: `PoolCommitter.setQuoteAndPool()` unncessarily approves `quoteToken`

[Severity: Low | Difficulty: -| Category: Implementation flaws]

The `setQuoteAndPool()` function of the `PoolCommitter` contract is used by the `PoolFactory` to initialize the pool committer. In this function, the committer approves the pool to spend an unlimited amount of settlement tokens on its behalf. However, all the internal balances and collaterals used to operate the pools are owned by the pool and not the committer. Consequently, the approval is superfluous.

Status

Acknowledged by the client.

B14: `PriceObserver` - potential gas optimization

[Severity: - | Difficulty: - | Category: Gas optimization]

The complexity of `leftRotateWithPad()` is `O(n)`, where it is called whenever a new price is added. The new price addition can be reduced to `O(1)` by employing a circular structure, that is, that `get(i)` returns `observations[(pos + i) % n]` where `pos` increases each time a new element is added. It needs more implementation effort and we recommend a follow-up audit, but the gas-saving will be significant considering the frequency of price updates as well as the current high gas price.

Status

Waiting for the client's feedback.

B15: `PriceObserver` - avoid magic values

[Severity: Low | Difficulty: - | Category: Code readability]

The magic value 24 is better to be replaced by the constant MAX_NUM_ELEMS in PriceObserver.sol, SMAOracle.sol, and IPriceObserver.sol. Having a single source of truth for constants lowers the risk of forgetting to update an occurrence when the value needs to be changed. Also, it gives a self-explanatory name to the otherwise anonymous value.

Status

Acknowledged by the client.

B16: `PoolCommitter` - Simplify burning logic

[Severity: Mid | Difficulty: - | Category: Code readability]

In `applyCommitment()`, for the `LongBurn` type commit, the `userAggregateBalance[msg.sender].longTokens` could be immediately updated, instead of having the extra bookkeeping logic using `userCommit.balanceLongBurnAmount` and later updating the user balance in `updateAggregateBalance()`. Similarly, the other burn type commits (`ShortBurn`, `LongBurnShortMint`, and `ShortBurnLongMint`) could be simplified as such. This allows for removing the extra bookkeeping logic, making the overall logic simpler, improving the code readability, and reducing gas cost.

Status

Acknowledged by the client.

B17: `PoolCommitter` - Simplify virtual balancing logic

[Severity: Mid | Difficulty: - | Category: Code readability]

Instead of burning tokens at the commitment time, they could be temporarily transferred to the pool and burned later when the commitment is actually executed. This allows for eliminating the virtual balance logic, which is highly non-trivial and difficult to reason about as witnessed by A08.

Status

Acknowledged by the client.

B18: `PoolCommitter` - Reduce code duplication

[Severity: Low | Difficulty: - | Category: Code readability]

The code duplication between `getAggregateBalance()` and `updateAggregateBalance()` could be mitigated by factoring out the common part. This will help to avoid any potential mistakes when updating the common part in the future.

Status

Acknowledged by the client.

B19: `PoolCommitter` - Improve naming conventions

[Severity: Low | Difficulty: - | Category: Code readability]

The PoolCommitter contract interacts with three different tokens: the settlement tokens, the long tokens, and the short tokens. The settlement tokens are transferred between the user of the contract and the pool. The long-, and short-tokens can be minted or burned in exchange for collateral. Whenever tokens are transferred, minted, or burned it would benefit readability if the name of the variables signaled the direction of the flow, for instance by suffixing them with `In` or `Out`.

Status

Acknowledged by the client.