Tracer Security Review

SPEARBIT

Reviewers

Gerard Persoon, Lead Security Researcher

Christoph Michel, Lead Security Researcher

Emanuele Ricci, Security Researcher

Rusty (f7dd60e9cfad19996d73), Security Researcher

April 17, 2022

1 Executive Summary

Over the course of 10 days in total, Tracer engaged with Spearbit to review Tracer Perpetual Pools.

We found a total of 51 issues with Tracer.

Repository	Commit
Tracer Perpetual Pools V2	3dc3e2202c9767275905d152eb2ed8bae6471141

Summary

Type of Project	Perpetual Swaps, DeFi
Timeline	Feb 1st - Feb 14th
Methods	Manual Review
Documentation	Medium
Testing Coverage	Medium-High

Total Issues

Critical Risk	1
High Risk	5
Medium Risk	9
Low Risk	6
Gas Optimizations	18
Informational	12

Contents

2 Spearbit

Spearbit is a decentralized network of expert Web3 security engineers. Together, we help secure the Web3 ecosystem. We offer security reviews and related services to Web3 projects. Our network has experience at every part of the stack, including protocol design, smart contracts, and the Solidity compiler itself. Spearbit brings in untapped security talent: expert freelance auditors who want flexibility to work on interesting projects together. Learn more about us at https://spearbit.com.

3 Introduction

TracerDAO introduces a derivative infrastructure called Perpetual Pools. Users create leveraged tokens, whereby long and short positions have changing claims on a pool of collateral. As long and short collateral tends to parity due to the implicit rebalancing rate, users' returns are expected to simulate a constantly rebalancing leveraged position. These positions cannot be liquidated, are fully collateralised, and can exist perpetually without upkeep. Several improvements and new features have been implemented in a second version (V2) of their contracts.

The focus of the security review has been on the following:

- Detecting general architecture vulnerabilities.
- Breaking and manipulating fee machanics.
- Abusing the `AutoClaim` functionality.
- Detecting incorrect protocol fee splits.
- Examining new keeper reward calculations.
- Stressing circuit breakers.

Disclaimer: This security review does not guarantee against a hack. It is a snapshot in time of Tracer Perpetual Pools according to the specific commit. Any modifications to the code will require a new security review.

4 Risk classification

Severity level	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: high	Critical	High	Medium
Likelihood: medium	High	Medium	Low
Likelihood: low	Medium	Low	Low

4.1 Impact

- High - leads to a loss of a significant portion (>10%) of assets in the protocol, or significant harm to a majority of users
- Medium - global losses <10% or losses to only a subset of users, but still unacceptable
- Low - losses will be annoying but bearable--applies to things like griefing attacks that can be easily repaired or even gas inefficiencies

4.2 Likelihood

- High - almost certain to happen, easy to perform, or not easy but highly incentivized
- Medium - only conditionally possible or incentivized, but still relatively likely
- Low - requires stars to align, or little-to-no incentive

4.3 Action required for severity levels

- Critical - Must fix as soon as possible (if already deployed)
- High - Must fix (before deployment if not already deployed)
- Medium - Should fix
- Low - Could fix

5 Findings

5.1 Critical Risk

5.1.1 Pool token price is incorrect when there is more than one pending upkeep

Severity: Critical

Context: PoolCommitter.sol#L384-391

Description: The amount of pool tokens to mint and quote tokens to burn is determined by the pool token price. This price, for a commit at update interval ID `X`, should not be influenced by any pending commits for IDs greater than `X`.

However, in the current implementation price includes the *current total supply* but `burn` commits burn pool tokens immediately when `commit()` is called, not when `upkeep()` is executed.

```
// pool token price computation at execution of updateIntervalId, example for long price
priceHistory[updateIntervalId].longPrice = longBalance /
    (IERC20(tokens[LONG_INDEX]).totalSupply() + _totalCommit[updateIntervalId].longBurnAmount +
↪   _totalCommit[updateIntervalId].longBurnShortMintAmount)
```

The implementation tries to fix this by adding back all tokens burned at this `updateIntervalId` but it must also add back all tokens that were burned in future commits (i.e. when `ID > updateIntervalID`).

This issue allows an attacker to get a better pool token price and steal pool token funds.

Example: Given the preconditions:

- `long.totalSupply() = 2000`
- User owns `1000` long pool tokens
- `lastPriceTimestamp = 100`
- `updateInterval = 10`
- `frontRunningInterval = 5`

At time `104`: User commits to `BurnLong` 500 tokens in `appropriateUpdateIntervalId = 5`. Upon execution user receives a long price of `longBalance / (1500 + 500)` if no further *future* commitments are made. Then, as tokens are burned `totalPoolCommitments[5].longBurnAmount = 500` and `long.totalSupply -= 500`.

At time `106`: User commits another 500 tokens to `BurnLong` at `appropriateUpdateIntervalId = 6` as they are now past the `frontRunningInterval` and are scheduled for the next update. Now `totalPoolCommitments[6].longBurnAmount = 500`, `long.totalSupply -= 500` again as tokens are burned.

Finally, the 5th update interval ID is executed by the pool keeper but at `longPrice = longBalance / (IERC20(tokens[LONG_INDEX]).totalSupply() + _totalCommit[5].longBurnAmount + _totalCommit[5].longBurnShortMintAmount = longBalance / (1000 + 500)` which is a better price than what the user should have received.

With a `longBalance` of `2000`, the user receives `500 * (2000 / 1500) = 666.67` tokens executing the first burn commit and `500 * ((2000 - 666.67) / 1500) = 444.43` tokens executing the second one.

The total pool balance received by the user is `1111.1/2000 = 55.555%` by burning only `1000 / 2000 = 50%` of the pool token supply.

Recommendation: Pool price computation should take into account all tokens that have been burned, not only the tokens that have been burned in the `updateIntervalID` of the commit. Note that there can be many pending total commits if `frontRunningInterval > updateInterval`.

Tracer: Valid. Fixed in commit 669a61a.

Spearbit: Acknowledged.

5.2 High Risk

5.2.1 No price scaling in SMAOracle

Severity: High Risk

Context: SMAOracle.sol#L82-L96, ChainlinkOracleWrapper.sol#L36-L60

Description: The `update()` function of the `SMAOracle` contract doesn't scale the `latestPrice` although a `scaler` is set in the constructor. On the other hand, the `_latestRoundData()` function of `ChainlinkOracleWrapper` contract does scale via `toWad()`.

```
contract SMAOracle is IOracleWrapper {

    constructor(..., uint256 _spotDecimals, ...) {
        ...
        require(_spotDecimals <= MAX_DECIMALS, "SMA: Decimal precision too high");
        ...
        /* `scaler` is always <= 10^18 and >= 1 so this cast is safe */
        scaler = int256(10**(MAX_DECIMALS - _spotDecimals));
        ...
    }
    function update() internal returns (int256) {
        /* query the underlying spot price oracle */
        IOracleWrapper spotOracle = IOracleWrapper(oracle);
        int256 latestPrice = spotOracle.getPrice();
        ...
        priceObserver.add(latestPrice);   // doesn't scale latestPrice
        ...
}
```

```
contract ChainlinkOracleWrapper is IOracleWrapper {
    function getPrice() external view override returns (int256) {
        (int256 _price, ) = _latestRoundData();
        return _price;
    }
    function _latestRoundData() internal view returns (int256, uint80) {
        (..., int256 price, ..) = AggregatorV2V3Interface(oracle).latestRoundData();
        ...
        return (toWad(price), ...);
    }
```

Recommendation: The `latestPrice` variable in `SMAOracle` contract should be scaled, and the `toWad()` function should be re-introduced.

Note: If the `SMAOracle` is only used with WAD based spot oracles, then `_spotDecimals == 18` must be enforced.

Tracer: We are submitting PR 406 as a mitigation for this. It is a slightly larger PR than we originally intended so as a result it will likely be submitted for several defects here. We would appreciate if each defect could be assessed against it.

Spearbit: Acknowledged.

5.2.2 Two different invariantCheck variables used in PoolFactory.deployPool()

Severity: High Risk

Context: PoolFactory.sol#L93-L174, IPoolFactory.sol#L14

Description: The deployPool() function in the PoolFactory contract uses two different invariantCheck variables: the one defined as a contract's instance variable and the one supplied as a parameter.

Note: This was also documented in Secureum's CARE-X report issue "Invariant check incorrectly fixed".

```
function deployPool(PoolDeployment calldata deploymentParameters) external override returns (address) {
        ...
        poolCommitter.initialize(...,  deploymentParameters.invariantCheck, ... );  // version 1 of
↪   invariantCheck
        ...
        ILeveragedPool.Initialization memory initialization = ILeveragedPool.Initialization({
            ...
            _invariantCheckContract: invariantCheck,  // version 2 of invariantCheck
            ...
        });
```

Recommendation: The code should be changed to:

```
function deployPool(PoolDeployment calldata deploymentParameters) external override returns (address) {
        ...
-        poolCommitter.initialize(...,  deploymentParameters.invariantCheck, ... );
+        poolCommitter.initialize(...,  invariantCheck, ... );
}
```

In addition, the invariantCheck member of struct PoolDeployment in IPoolFactory.sol should be removed to prevent mistakes.

Tracer: Valid. Fixed as part of CARE-X commit 98c76bf

Spearbit: Acknowledged.

5.2.3 Duplicate user payments for long commits when paid from balance

Severity: High Risk

Context: PoolCommitter.sol#L299-L306

Description: When minting pool tokens in commit(), the fromAggregateBalance parameter indicates if the user wants to pay from their internal balances or by transferring the tokens. The second if condition is wrong and leads to users having to pay twice when calling commit() with CommitType.LongMint and fromAggregateBalance = true.

Recommendation: The second if condition should be changed to only perform the transfer for pool token mints if they have not been already paid from internal balances.

```
-if (commitType == CommitType.LongMint || (commitType == CommitType.ShortMint &&
↪   !fromAggregateBalance)) {
+if ((commitType == CommitType.LongMint || commitType == CommitType.ShortMint) &&
↪   !fromAggregateBalance) {
    // minting: pull in the quote token from the committer
    // Do not need to transfer if minting using aggregate balance tokens, since the leveraged pool
    ↪   already owns these tokens.
    pool.quoteTokenTransferFrom(msg.sender, leveragedPool, amount);
}
```

Tracer: Already fixed in commit 4f2d38f

Spearbit: Previously the token transfer was done after the `applyCommitment()` probably to avoid re-entrancy issues. This behavior is different for non-ERC20 tokens such as ERC777 tokens that give control to the sender and recipient. Is the system intended to support these other token standards? Other than that, it is a valid fix.

Tracer: Our system only needs to support ERC20 tokens and our threat model encompasses this invariant.

Spearbit: Acknowledged.

5.2.4 Initial `executionPrice` is too high

Severity: High Risk

Context: PoolKeeper.sol#L73

Description: When a pool is deployed the initial `executionPrice` is calculated as `firstPrice * 1e18` where `firstPrice` is `ILeveragedPool(_poolAddress).getOraclePrice()`:

```
contract PoolKeeper is IPoolKeeper, Ownable {
    function newPool(address _poolAddress) external override onlyFactory {
        int256 firstPrice = ILeveragedPool(_poolAddress).getOraclePrice();
        int256 startingPrice = ABDKMathQuad.toInt(ABDKMathQuad.mul(ABDKMathQuad.fromInt(firstPrice),
↪   FIXED_POINT));
        executionPrice[_poolAddress] = startingPrice;
    }
}
```

All other updates to `executionPrice` use the result of `getPriceAndMetadata()` directly without scaling:

```
function performUpkeepSinglePool() {
   ...
   (int256 latestPrice, ...) = pool.getUpkeepInformation();
   ...
   executionPrice[_pool] = latestPrice;
   ...
}

contract LeveragedPool is ILeveragedPool, Initializable, IPausable {
    function getUpkeepInformation() {
        (int256 _latestPrice, ...) = IOracleWrapper(oracleWrapper).getPriceAndMetadata();
        return (_latestPrice, ...);
    }
}
```

The price after the `firstPrice` will always be lower, therefore its funding rate payment will always go to the shorts and long pool token holders will incur a loss.

Recommendation: The `1e18` scaling should be removed for the initial `executionPrice`

```
- int256 startingPrice = ABDKMathQuad.toInt(ABDKMathQuad.mul(ABDKMathQuad.fromInt(firstPrice),
↪   FIXED_POINT));
+ int256 startingPrice = firstPrice;
```

Tracer: Valid. Fixed in commit 445377f.

Spearbit: Acknowledged.

5.2.5 Paused state can't be set and therefore `withdrawQuote()` can't be executed

Severity: High Risk

Context: InvariantCheck, LeveragedPool, PoolCommitter

Description: The `checkInvariants()` function of the `InvariantCheck` contract is called via the modifiers `checkInvariantsBeforeFunction()` and `checkInvariantsAfterFunction()` of both `LeveragedPool` and `PoolCommitter` contracts, and it is meant to pause the contracts if the invariant checks don't hold.

The aforementioned modifiers also contain the `require(!paused, "Pool is paused");` statement, which reverts the entire transaction and resets the `paused` variable that was just set.

Furthermore, the `paused` state can only be set by the `InvariantCheck` contract due to the `onlyInvariantCheckContract` modifier. Thus the `paused` variable will never be set to `true`, making `withdrawQuote()` impossible to be executed because it requires the contract to be paused.

This means that the quote tokens will always stay in the pool even if invariants don't hold and all other actions are blocked.

Relevant parts of the code:

The `checkInvariants()` function calls `InvariantCheck.pause()` if the invariants don't hold. The latter calls `pause()` in `LeveragedPool` and `PoolCommitter`:

```
contract InvariantCheck is IInvariantCheck {
    function checkInvariants(address poolToCheck) external override {
        ...
        pause(IPausable(poolToCheck), IPausable(address(poolCommitter)));
        ...
    }
    function pause(IPausable pool, IPausable poolCommitter) internal {
          pool.pause();
          poolCommitter.pause();
     }
}
```

In `LeveragedPool` and `PoolCommitter` contracts, the `checkInvariantsBeforeFunction()` and `checkInvariantsAfterFunction()` modifiers will make the transaction revert if `checkInvariants()` sets the paused state.

```
contract LeveragedPool is ILeveragedPool, Initializable, IPausable {
    modifier checkInvariantsBeforeFunction() {
        invariantCheck.checkInvariants(address(this)); // can set paused to true
        require(!paused, "Pool is paused"); // will reset pause again
        _;
    }
    modifier checkInvariantsAfterFunction() {
        require(!paused, "Pool is paused");
        _;
        invariantCheck.checkInvariants(address(this)); // can set paused to true
        require(!paused, "Pool is paused"); // will reset pause again
    }
    function pause() external override onlyInvariantCheckContract {  // can only called from
↪   InvariantCheck
        paused = true;
        emit Paused();
    }
}
```

```
contract PoolCommitter is IPoolCommitter, Initializable {
    modifier checkInvariantsBeforeFunction() {
        invariantCheck.checkInvariants(leveragedPool); // can set paused to true
        require(!paused, "Pool is paused");  // will reset pause again
        _;
    }
    modifier checkInvariantsAfterFunction() {
        require(!paused, "Pool is paused");
        _;
        invariantCheck.checkInvariants(leveragedPool); // can set paused to true
        require(!paused, "Pool is paused"); // will reset pause again
    }
    function pause() external onlyInvariantCheckContract {  // can only called from InvariantCheck
        paused = true;
        emit Paused();
    }
```

Recommendation: This issue is also discussed in modifiers-undermine-contract-pausing and there are a few reasons to reconsider fixing it:

- When a transaction triggers an invariant check and it is rolled back due to the revert, other transactions might still be executed. With a paused state this wouldn't happen.
- Pause functionality that doesn't work is misleading for developers and code reviewers. It is better to delete the dead code, which also saves gas.
- `withdrawQuote()` cannot be executed since the pause state is unreachable.

If it is preferable to use the pause functionality, it can be done by having a surrounding contract which stores the pause state and calls the underlying logic with a `try` mechanism. The underlying logic can then be reverted while the surrounding contract keeps the `paused` state.

Here is an implementation example of a surrounding contract that handles the revert and sets the `pause` variable. The code that checks the invariants reverts with the `InvariantsFail()` custom error, which is caught by the `try/catch` block in the caller and the paused state is set.

```
//SPDX-License-Identifier: MIT
pragma solidity 0.8.11;
import "hardhat/console.sol";

contract Pool {
    error InvariantsFail();
    function DoSomething() public pure {
        revert InvariantsFail();
    }
}
contract InvariantCheck {
    Pool myPool = new Pool();
    bool pause;
    function TryDoSomething() public returns (string memory) {
        try myPool.DoSomething()          { return "Ok"; }
        catch Error(string memory reason) { return reason; }
        catch Panic(uint)                 { return "Panic"; }
        catch (bytes memory reason) {
            if (bytes4(reason) == bytes4(abi.encodeWithSignature("InvariantsFail()"))) {
                pause = true;
                return("InvariantsFail");
            }
            return "Unknown";
        }
    }
    constructor() {
        console.log("Pause =",pause);
        console.log(TryDoSomething());
        console.log("Pause =",pause);
    }
}
```

Note: Beware that this does not interfere with any other functionality. ETH has to be sent back to the caller as this will not be automatically reverted.

Tracer: We decided to change how invariant checking works. Instead of checking the invariants on every function call, we now have a contract which can be called at any time by an EOA to do the same invariant checks/pausing.

This obviously does not have the same invariant guarantees, as it requires an EOA to start a TX to detect invariant violations, but we decided to make the tradeoff anyway. However, it does mean that this issue should not be relevant anymore, because the paused state can in fact be set.

Addressed in PR 384.

Spearbit: Acknowledged.

5.3 Medium Risk

5.3.1 The value of `lastExecutionPrice` fails to update if `pool.poolUpkeep()` reverts

Severity: Medium Risk

Context: PoolKeeper.sol#L119-L161

Description: The `performUpkeepSinglePool()` function of the `PoolKeeper` contract updates `executionPrice[]` with the latest price and calls `pool.poolUpkeep()` to process the price difference. However, `pool.poolUpkeep()` can revert, for example due to the `checkInvariantsBeforeFunction` modifier in `mintTokens()`.

If `pool.poolUpkeep()` reverts then the previous price value is lost and the processing will not be accurate. Therefore, it is safer to store the new price only if `pool.poolUpkeep()` has been executed succesfully.

```
function performUpkeepSinglePool(...) public override {
        ...
        int256 lastExecutionPrice = executionPrice[_pool];
        executionPrice[_pool] = latestPrice;   // previous price can get lost if poolUpkeep() reverts
        ...
        try pool.poolUpkeep(lastExecutionPrice, latestPrice, _boundedIntervals, _numberOfIntervals) {
          ...   // executionPrice[_pool] should be updated here
        } catch Error(string memory reason) {
            ...
        }
    }
```

Recommendation: To prevent losing the `latestPrice` value, the `executionPrice[_pool]` variable should be updated only if `poolUpkeep()` doesn't revert.

```
 function performUpkeepSinglePool(...) public override {
        ...
        int256 lastExecutionPrice = executionPrice[_pool];
-       executionPrice[_pool] = latestPrice;
        ...
        try pool.poolUpkeep(lastExecutionPrice, latestPrice, _boundedIntervals, _numberOfIntervals) {
+           executionPrice[_pool] = latestPrice;
            ...
        } catch Error(string memory reason) {
            ...
        }
    }
```

Alternatively, the administration of the `executionPrice`s could be done within the called `poolUpkeep()` function of the `LeveragedPool` contract.

Tracer: Valid, fixed in PR 327.

Spearbit: Acknowledged.

5.3.2 Pools can be deployed with malicious or incorrect quote tokens and oracles

Severity: Medium Risk

Context: PoolFactory.sol#L93-L174

Description: The deployment of a pool via `deployPool()` is permissionless. The deployer provides several parameters that have to be trusted by the users of a specific pool, these parameters include:

- `oracleWrapper`
- `settlementEthOracle`
- `quoteToken`
- `invariantCheck`

If any one of them is malicious, then the pool and its value will be affected.

Note: Separate findings are made for the deployer check (issue *Authenticity check for oracles is not effective*) and the `invariantCheck` (issue *Two different `invariantCheck` variables used in `PoolFactory.deployPool()`*).

Recommendation: Although this is a general risk with permissionless protocols, it is possible to add extra controls (such as allowlists) on quote tokens and the corresponding `settlementEthOracle` oracle. An additional benefit of allowlisting quote tokens and corresponding oracles is that the oracles could be shared, thus saving gas.

Tracer: As decided by the core Tracer team, no allowlists are going to be added.

We do agree, though, that there is a risk. To address it without impinging on any degree of permissionlessness, the DAO will be carrying out security checks to give a safety score (akin to Rari protocol's security checks).

Spearbit: Acknowledged.

5.3.3 `pairTokenBase` and `poolBase` template contracts instances are not initialized

Severity: Medium Risk

Context: PoolFactory.sol#L66-L81, PoolToken.sol#L9-L14, ERC20_Cloneable.sol#L33-L72, LeveragedPool.sol#L90-L133

Description: The `constructor` of `PoolFactory` contract creates three template contract instances but only one is initialized: `poolCommitterBase`. The other two contract instances (`pairTokenBase` and `poolBase`) are not initialized.

```
contract PoolFactory is IPoolFactory, Ownable {
   constructor(address _feeReceiver) {
        ...
        PoolToken pairTokenBase = new PoolToken(DEFAULT_NUM_DECIMALS); // not initialized
        pairTokenBaseAddress = address(pairTokenBase);
        LeveragedPool poolBase = new LeveragedPool(); // not initialized
        poolBaseAddress = address(poolBase);
        PoolCommitter poolCommitterBase = new PoolCommitter(); // is initialized
        poolCommitterBaseAddress = address(poolCommitterBase);
        ...
        /* initialise base PoolCommitter template (with dummy values) */
        poolCommitterBase.initialize(address(this), address(this), address(this), owner(), 0, 0, 0);
    }
```

This means an attacker can initialize the templates setting them as the owner, and perform `owner` actions on contracts such as minting tokens. This can be misleading for users of the protocol as these minted tokens seem to be valid tokens.

In `PoolToken.initialize()` an attacker can become the owner by calling `initialize()` with an address under his control as a parameter. The same can happen in `LeveragedPool.initialize()` with the `initialization` parameter.

```
contract PoolToken is ERC20_Cloneable, IPoolToken {
    ...
}
contract ERC20_Cloneable is ERC20, Initializable {
    function initialize(address _pool, ) external initializer { // not called for the template contract
        owner = _pool;
        ...
    }
}
```

```
contract LeveragedPool is ILeveragedPool, Initializable, IPausable {
    function initialize(ILeveragedPool.Initialization calldata initialization) external override
↪   initializer {
        // not called for the template contract
        ...
        // set the owner of the pool. This is governance when deployed from the factory
        governance = initialization._owner;
    }
}
```

Recommendation: `pairTokenBase` and `poolBase` should be initialized with dummy values.

Consider using the upgradable versions of the OpenZeppelin ERC20 contracts, as they don't have constructors.

Tracer: Valid, fixed in PR 396.

Spearbit: Acknowledged.

5.3.4 Oracles are not updated before use

Severity: Medium Risk

Context: PoolKeeper.sol#L71, PoolKeeper.sol#L281

Description: The `PoolKeeper` contract uses two oracles but does not ensure that their prices are updated. The `poll()` function should be called on both oracles to get the first execution and the settlement / ETH prices. As it currently is, the code could operate on old data.

Recommendation: Price should be updated before using it. Note that calling `poll()` can revert in certain cases depending on the oracle type if:

1. It does not have enough data to create an SMA
2. Not enough time has passed since it was last updated (`require(block.timestamp >= lastUpdate + updateInterval, "SMA: Too early to update")`)

Therefore, it is recommended to add a `try {} catch {}` block as shown in the code below:

```
function newPool(address _poolAddress) external override onlyFactory {
+    try IOracleWrapper(ILeveragedPool(_poolAddress).oracleWrapper()).poll() {} catch Error(string
↪   memory reason) {}
    int256 firstPrice = ILeveragedPool(_poolAddress).getOraclePrice();
    ...
}

function performUpkeepSinglePool(...) public override {
    ...
    try pool.poolUpkeep(lastExecutionPrice, latestPrice, _boundedIntervals, _numberOfIntervals) {
        ...
+       try IOracleWrapper(ILeveragedPool(_pool).settlementEthOracle()).poll() {} catch Error(string
↪   memory reason) {}
        payKeeper(_pool, gasPrice, gasSpent, savedPreviousUpdatedTimestamp, updateInterval);
        ...
    } catch Error(string memory reason) {
        ...
    }
}
```

Tracer: Regarding the `settlementEthOracle`, this should be a spot oracle, in which case `poll()` is a no-op. We are not sure if we should then still include a call to its `poll()` function.

For the pool's main Oracle wrapper, it is OK to not call the `poll()` function when the pool is added, because it is polled right when the pool upkeeps. This means the latest poll will get the most up-to-date price. We also now have a ramping-up feature in the SMA oracle when it is populated fewer times than the total number of periods available.

Spearbit:

> *Regarding the `settlementEthOracle`, this should be a spot oracle, in which case `poll()` is a no-op. We are not sure if we should then still include a call to its `poll()` function.*

If spot oracle means `ChainlinkOracleWrapper` contract instead of `SMAOracle`, calling `poll()` for it does not change its state because it is the same as `getPrice()`.

However, the `settlementOracle` is chosen by the deployer and people might as well deploy a `SMAOracle`, even if you intend it to be used differently, because your intent is not documented anywhere. In this case, it is better to trigger a `poll()`, or document that it should be a spot oracle.

> *For the pool's main oracle wrapper, it is OK to not call the `poll()` function when the pool is added, because it is polled right when the pool upkeeps, meaning the latest poll will be the most up-to-date price update.*

We agree that you are already calling `poll()` on upkeep. This is about calling `poll()` for the `startingPrice` in `newPool`, so the pool's start price is the latest one.

Tracer:

> *We agree that you are already calling `poll()` on upkeep. This is about calling `poll()` for the `startingPrice` in `newPool`, so the pool's start price is the latest one*

Good point.

Spearbit: The recommendation has been implemented in PR 400 with the addition that if the `poll` fails, it will also emit a `PoolUpkeepError`.

5.3.5 `getPendingCommits()` underreports commits

Severity: Medium Risk

Context: PoolCommitter.sol#L764

Description: When `frontRunningInterval > updateInterval`, the `PoolCommitter.getAppropriateUpdateIntervalId()` function can return `updateInterval` IDs that are arbitrarily far into the future, especially if `appropriateIntervalId > updateIntervalId + 1`.

Therefore, commits can also be made to these appropriate interval IDs far in the future by calling `commit()`. The `PoolCommitter.getPendingCommits()` function only checks the commits for `updateIntervalId` and `updateIntervalId + 1`, but needs to check up to `updateIntervalId + factorDifference + 1`.

Currently, it is underreporting the pending commits which leads to the `checkInvariants` function not checking the correct values.

Recommendation: The `getPendingCommits` function should return all possible pending commits even in the case where `frontRunningInterval > updateInterval`.

Tracer: As part of the CARE program, we found that `getPendingCommits()` can be removed in favour of a running total of pending mints. This was done and merged into the main repository in PR 315

Spearbit: Fix looks good but naming the variable `totalPendingMints` is a bit ambiguous because:

- It sounds like it is a pool token amount but it is a quote token amount. However, the naming for other vars (`longMintAmount` etc.) never distinguished this either, so at least it is consistent.
- It does not include mints from `shortBurnLongMint/longBurnShortMint` which are also mints. It does not include these because you do not need to track it for the invariant checks. You could add a remark here about the exclusion

Tracer: Addressed in issue 368 and PR 403.

Spearbit: Variable names were refactored in PR 403, to indicate if they are in settlement tokens or pool tokens.

5.3.6 Authenticity check for oracles is not effective

Severity: Medium Risk

Context: PoolFactory.sol#L93-L101

Description: The `deployPool()` function verifies the authenticity of the `oracleWrapper` by calling its `deployer()` function. As the `oracleWrapper` is supplied via `deploymentParameters`, it can be a malicious contract whose `deployer()` function can return any value, including `msg.sender`.

Note: this check does protect against frontrunning the deployment transaction of the same pool. See *Undocumented frontrunning protection*.

```
function deployPool(PoolDeployment calldata deploymentParameters) external override returns (address) {
        ...
        require(IOracleWrapper(deploymentParameters.oracleWrapper).deployer() == msg.sender,
            "Deployer must be oracle wrapper owner");
```

Recommendation: Consider using an allowlist for oracles.

Tracer: As decided by the core Tracer team, no allowlists are going to be added.

We do agree, though, that there is a risk. To address it without impinging on any degree of permissionlessness, the DAO will be carrying out security checks to give a safety score (akin to Rari protocol's security checks).

Spearbit: Acceptable if it is clearly shown in the user interface.

5.3.7 Incorrect calculation of keeper reward

Severity: Medium Risk

Context: PoolKeeper.sol#L251-L259

Description: The keeper reward is calculated as `(keeperGas * tipPercent / 100) / 1e18`. The division by `1e18` is incorrect and undervalues the reward for the keeper. The tip part of the keeper reward is essentially ignored.

The likely cause of this miscalculation is based on the note at PoolKeeper.sol#244 which states the tip percent is in WAD units, but it really is a quad representation of a value in the range between `5` and `100`.

The comment at PoolKeeper.sol#L241 also incorrectly states that `_keeperGas` is in `wei` (usually referring to ETH), which is not the case as it is denominated in the quote token, but in WAD precision.

Recommendation: The division by `FIXED_POINT` should be removed, and the comments should be corrected.

```
int256 wadRewardValue = ABDKMathQuad.toInt(
    ABDKMathQuad.add(
        ABDKMathQuad.fromUInt(_keeperGas),
-        ABDKMathQuad.div(
            (
                ABDKMathQuad.div(
                    (ABDKMathQuad.mul(ABDKMathQuad.fromUInt(_keeperGas), _tipPercent)),
                    ABDKMathQuad.fromUInt(100)
                )
            ),
-            FIXED_POINT
-        )
    )
);
```

Tracer: Valid, addressed in PR 391 and PR 428.

Spearbit: Acknowledged.

5.3.8 `performUpkeepSinglePool()` can result in a griefing attack when the pool has not been updated for many intervals

Severity: Medium Risk

Context: executeCommitments() in PoolCommitter

Description: Assuming the pool has not been updated for many update intervals, `performUpkeepSinglePool()` can call `poolUpkeep()` repeatedly with `_boundedIntervals == true` and a bounded amount of gas to fix this situation. This in turn will call `executeCommitments()` repeatedly.

For each call to `executeCommitments()` the `updateMintingFee()` function will be called. This updates fees and changes them in an unexpected way. A griefing attack is possible by repeatedly calling `executeCommitments()` with `boundedIntervals == true` and `numberOfIntervals == 0`.

Note: Also see issue *It is not possible to call `executeCommitments()` for multiple old commits*. It is also important that `lastPriceTimestamp` is only updated after the last `executeCommitments()`, otherwise it will revert.

```
function executeCommitments(bool boundedIntervals, uint256 numberOfIntervals) external override
↪   onlyPool {
    ...
    uint256 upperBound = boundedIntervals ? numberOfIntervals : type(uint256).max;
    ...
        while (i < upperBound) {
            if (block.timestamp >= lastPriceTimestamp + updateInterval * counter) {  //
↪   lastPriceTimestamp  shouldn't be updated too soon
                ...
        }
    }
    ...
    updateMintingFee(); // should do this once (in combination with _boundedIntervals==true)
    ...
}
```

Recommendation: Ensure `updateMintingFee()` is only called once in a series of calls to `executeCommitments()` with `_boundedIntervals == true`.

Tracer: We want the minting fee to be the most up-to-date whenever anyone commits to a mint. This means we do not need to update it every iteration, but we do want to make sure it is the most up-to-date at the end of the commitment executions.

In the case of not executing all update intervals, we think it makes sense to still update the minting fee, so if someone commits to a mint between this call to `executeCommitments()` and the next, they at least have a minting fee that has been updated since the last update interval.

Addressed in PR 413.

Spearbit: Acknowledged.

5.3.9 It is not possible to call `executeCommitments()` for multiple old commits

Severity: Medium Risk

Context: poolUpkeep() in LeveragedPool

Description: Assuming the pool has not been updated for many update intervals, `performUpkeepSinglePool()` can call `poolUpkeep()` repeatedly with `_boundedIntervals == true` and a bounded amount of gas to fix this situation.

In this context the following problem occurs:

- In the first run of `poolUpkeep()`, `lastPriceTimestamp` will be set to `block.timestamp`.
- In the next run of `poolUpkeep()`, processing will stop at `require(intervalPassed(),..)`, because `block.timestamp` hasn't increased.

This means the rest of the commitments won't be executed by `executeCommitments()` and `updateIntervalId`, which is updated in `executeCommitments()`, will start lagging.

```
  function poolUpkeep(..., bool _boundedIntervals, uint256 _numberOfIntervals) external override
↪  onlyKeeper {
        require(intervalPassed(), "Update interval hasn't passed"); // next time lastPriceTimestamp ==
↪  block.timestamp
        executePriceChange(_oldPrice, _newPrice); // should only to this once (in combination with
↪  _boundedIntervals==true)
        IPoolCommitter(poolCommitter).executeCommitments(_boundedIntervals, _numberOfIntervals);
        lastPriceTimestamp = block.timestamp; // shouldn't update until all executeCommitments() are
↪  processed
    }
 function intervalPassed() public view override returns (bool) {
        unchecked {
            return block.timestamp >= lastPriceTimestamp + updateInterval;
        }
    }
}
```

Recommendation:

- Redesign the logic with `boundedIntervals` (see the related issues)
- Update `lastPriceTimestamp` only once all old commitments are processed.
- Ensure `executePriceChange()` is only executed once for a series of old commitments to prevent negative side effects.

Tracer: As part of PR 392, we have a hardcoded limit to avoid running out of gas, but this also solves the user-supplied data problems. Let us know if you think this is a valid solution.

Spearbit: Acknowledged.

5.4 Low Risk

5.4.1 Incorrect comparison in `getUpdatedAggregateBalance()`

Severity: Low Risk

Context: PoolSwapLibrary.sol#L476-L490

Description: When the value of `data.updateIntervalId` accidentally happens to be larger than `data.currentUpdateIntervalId` in the `getUpdatedAggregateBalance()` function, it will execute the rest of the function, which shouldn't happen. Although this is unlikely it is also very easy to prevent.

```
 function getUpdatedAggregateBalance(UpdateData calldata data) external pure returns (...) {
        if (data.updateIntervalId == data.currentUpdateIntervalId) {
            // Update interval has not passed: No change
            return (0, 0, 0, 0, 0);
        }
}
```

Recommendation: The code should be changed to:

```
 function getUpdatedAggregateBalance(UpdateData calldata data)  external pure returns (...) {
-       if (data.updateIntervalId == data.currentUpdateIntervalId) {
+       if (data.updateIntervalId >= data.currentUpdateIntervalId) {
            // Update interval has not passed: No change
            return (0, 0, 0, 0, 0);
        }
}
```

Tracer: Valid, fixed in commit 259386d.

Spearbit: Acknowledged.

5.4.2 `updateAggregateBalance()` can run out of gas

Severity: Low Risk

Context: PoolCommitter.sol#L609-L682

Description: The `updateAggregateBalance()` function of the `PoolCommitter` contract contains a `for` loop that, in theory, could use up all the gas and result in a revert.

The `updateAggregateBalance()` function checks all future intervals every time it is called and adds them back to the `unAggregatedCommitments` array, which is checked in the next function call. This would only be a problem if `frontRunningInterval` is much larger than `updateInterval`, a situation that seems unlikely in practice.

```
    function updateAggregateBalance(address user) public override checkInvariantsAfterFunction {
        ...
        uint256[] memory currentIntervalIds = unAggregatedCommitments[user];
        uint256 unAggregatedLength = currentIntervalIds.length;
        for (uint256 i = 0; i < unAggregatedLength; i++) {
            uint256 id = currentIntervalIds[i];
            ...
            UserCommitment memory commitment = userCommitments[user][id];
            ...
            if (commitment.updateIntervalId < updateIntervalId) {
               ...
            } else {
              ...
              storageArrayPlaceHolder.push(currentIntervalIds[i]); // entry for future intervals stays
↪   in array
            }
        }
        delete unAggregatedCommitments[user];
        unAggregatedCommitments[user] = storageArrayPlaceHolder;
        ...
    }
```

Recommendation: An upper limit to the number of future intervals (e.g. `frontRunningInterval` / `updateInterval`) should be set in the `initialize()` function of `LeveragedPool` contract.

Tracer: Addressed in PR 392.

Spearbit: Have you checked `MAX_ITERATIONS = type(uint8).max` loops is possible within gas limits?

`updateAggregateBalance()` deletes all `unAggregatedCommitments[]` while there may be some commitments that have not been processed if the limit was reached. `getAggregateBalance()` does limit the loop so it could give a different result.

Tracer: Thanks for raising that. We created this PR addressing it. We also realised `PoolSwapLibrary::appropriateUpdateIntervalId` was still buggy when the frontrunning interval is greater than the update interval.

Spearbit: It looks good. Some minor suggestions:

`commitmentIds.pop()` can be put after the `if` statement, as it is executed both in the `if` and `else` blocks.

It is safer to put `commitmentIds.length > 1` before the `i < commitmentIds.length - 1`. If `commitmentIds.length` happens to be `0` then the statement will revert at `commitmentIds.length - 1`, although with the current code this will not happen.

```
if (unAggregatedLength > MAX_ITERATIONS && i < commitmentIds.length - 1 && commitmentIds.length > 1) {
    commitmentIds[i] = commitmentIds[commitmentIds.length - 1];
    commitmentIds.pop();
} else {
    commitmentIds.pop();
}
```

Tracer: We implemented the changes in PR 430.

Spearbit: Acknowledged.

5.4.3 Pool information might be lost if `setFactory()` of `PoolKeeper` contract is called

Severity: Low Risk

Context: PoolKeeper.sol#L324-L327, PoolKeeper.sol#L83-L86

Description: The `PoolKeeper` contract has a function to change the factory: `setFactory()`. However, calling this function will make previous pools inaccessible for this `PoolKeeper` unless the new factory imports the pools from the old factory.

The `isUpkeepRequiredSinglePool()` function calls `factory.isValidPool(_pool)`, and it will fail because the new factory doesn't know about the old pools. As this call is essential for upkeeping, the entire upkeep mechanism will fail.

```
function setFactory(address _factory) external override onlyOwner {
       factory = IPoolFactory(_factory);
       ...
   }

function isUpkeepRequiredSinglePool(address _pool) public view override returns (bool) {
       if (!factory.isValidPool(_pool)) {   // might not work if factory is changed
           return false;
       }
       ...
   }
```

Recommendation: Make sure the implementations of `setFactory()` and `isUpkeepRequiredSinglePool()` are correct to specification when the factory is changed.

Tracer: Valid, fixed in PR 340.

Spearbit: Acknowledged.

5.4.4 Ether could be lost when calling `commit()`

Severity: Low Risk

Context: PoolCommitter.sol#L263-L317

Description: The `commit()` function sends the supplied ETH to `makePaidClaimRequest()` only if `payForClaim == true`. If the caller of `commit()` accidentally sends ETH when `payForClaim == false` then the ETH stays in the `PoolCommitter` contract and is effectively lost.

Note: This was also documented in Secureum's CARE Tracking

```
function commit(...) external payable override checkInvariantsAfterFunction {
      ...
       if (payForClaim) {
           autoClaim.makePaidClaimRequest{value: msg.value}(msg.sender);
       }
   }
```

Recommendation: Consider changing the code to:

```
 function commit(...) external payable override checkInvariantsAfterFunction {
       ...
      if (payForClaim) {
+        require(msg.value != 0, "Must pay for claim");
         autoClaim.makePaidClaimRequest{value: msg.value}(msg.sender);
      } else {
+        require(msg.value == 0, "user's ETH would be lost");
      }
}
```

Tracer: Valid, fixed in PR 326.

Spearbit: Acknowledged.

5.4.5 Race condition if `PoolFactory` deploy pools before fees are set

Severity: Low Risk

Context: PoolFactory, PoolCommitter

Description: The `deployPool` function of `PoolFactory` contract can deploy pools before the `changeInterval` value and minting and burning fees are set. This means that fees would not be subtracted.

The exact boundaries for the `mintingFee`, `burningFee` and `changeInterval` values aren't clear. In some parts of the code `< 1e18` is used, and in other parts `<= 1e18`.

Furthermore, the `initialize()` function of the `PoolCommitter` contract doesn't check the value of `changeInterval`. The `setBurningFee()`, `setMintingFee()` and `setChangeInterval()` functions of `PoolCommitter` contract don't check the new values.

Finally, two representations of `1e18` are used: `1e18` and `PoolSwapLibrary.WAD_PRECISION`.

```
contract PoolFactory is IPoolFactory, Ownable {
   function setMintAndBurnFeeAndChangeInterval(uint256 _mintingFee, uint256 _burningFee,...) ... {
        ...
        require(_mintingFee <= 1e18, "Fee cannot be > 100%");
        require(_burningFee <= 1e18, "Fee cannot be > 100%");
        require(_changeInterval <= 1e18, "Change interval cannot be > 100%");
        mintingFee = _mintingFee;
        burningFee = _burningFee;
        changeInterval = _changeInterval;
        ...
    }
    function deployPool(PoolDeployment calldata deploymentParameters) external override returns
↪   (address) {
        ... // no check that mintingFee, burningFee, changeInterval are set
        poolCommitter.initialize(...,  mintingFee, burningFee, changeInterval, ...);
   }
}
```

```
contract PoolCommitter is IPoolCommitter, Initializable {

   function initialize(... ,uint256 _mintingFee, uint256 _burningFee,... ) ... {
        ...
        require(_mintingFee < PoolSwapLibrary.WAD_PRECISION, "Minting fee >= 100%");
        require(_burningFee < PoolSwapLibrary.WAD_PRECISION, "Burning fee >= 100%");
        ... // no check on _changeInterval
        mintingFee = PoolSwapLibrary.convertUIntToDecimal(_mintingFee);
        burningFee = PoolSwapLibrary.convertUIntToDecimal(_burningFee);
        changeInterval = PoolSwapLibrary.convertUIntToDecimal(_changeInterval);
        ...
    }
    function setBurningFee(uint256 _burningFee) external override onlyGov {
        burningFee = PoolSwapLibrary.convertUIntToDecimal(_burningFee); // no check on _burningFee
        ...
    }
    function setMintingFee(uint256 _mintingFee) external override onlyGov {
        mintingFee = PoolSwapLibrary.convertUIntToDecimal(_mintingFee); // no check on _mintingFee
        ...
    }
    function setChangeInterval(uint256 _changeInterval) external override onlyGov {
        changeInterval = PoolSwapLibrary.convertUIntToDecimal(_changeInterval); // no check on
↪   _changeInterval
        ...
    }
    function updateMintingFee(bytes16 longTokenPrice, bytes16 shortTokenPrice) private {
            ...
            if (PoolSwapLibrary.compareDecimals(mintingFee, MAX_MINTING_FEE) == 1) {
                // mintingFee is greater than 1 (100%).
                // We want to cap this at a theoretical max of 100%
                mintingFee = MAX_MINTING_FEE;    // so mintingFee is allowed to be 1e18
            }
        }
}
```

Recommendation:

- Initialize the values of `mintingFee`, `burningFee` and `changeInterval` in the constructor of the `PoolFactory` contract, or check in the `deployPool()` function that the values for `mintingFee`, `burningFee`, and `changeInterval` are set.
- Double-check the maximum values of `mintingFee`, `burningFee` and `changeInterval`.
- Check the value of `_changeInterval` in the `initialize()` function of the `PoolCommitter` contract.
- Check the new values in `setBurningFee()`, `setMintingFee()` and `setChangeInterval()` functions of the `PoolCommitter` contract.
- Replace `1e18` with `PoolSwapLibrary.WAD_PRECISION`

Tracer: Disputed, as it is fine if we want 0 minting/burning fee and change interval. This is because the minting, burning, but most particularly the change interval, are things we want to experiment with in the real market and we want to be able to not use them if desired.

Furthermore the `PoolDeployment` type captures the market creator's desire for fees so this seems like a non-issue.

As for the bounds on both minting and burning fees, the inconsistency in enforcement of the upper bounds is a defect but we have since capped the burning fee arbitrarily at 10%, so this seems to be an implicit mitigation. We don't have a cap on what the minting fee can be. This was a decision made by our RnD.

Spearbit: It might be helpful to add a comment to the definition of the `mintingFee`, `burningFee` and `changeInterval` variables, stating that a zero-value is also allowed.

Tracer: Addressed in PR 421.

Spearbit: Acknowledged.

5.4.6 Committer not validated on withdraw claim and multi-paid claim

Severity: Low Risk

Context: AutoClaim.sol#L118, AutoClaim.sol#L136, AutoClaim.sol#L150

Description: `AutoClaim` checks that the committer creating the claim request in `makePaidClaimRequest` and withdrawing the claim request in `withdrawUserClaimRequest` is a valid committer for the `PoolFactory` used in the`AutoClaim` initializer.

The same security check should be done in all the other functions where the committer is passed as a function parameter.

Recommendation: A validity check should be added for the `poolCommitter` passed as a parameter, as it is done in scenarios where the `msg.sender` is the committer itself.

```
function multiPaidClaimMultiplePoolCommitters(address[] calldata users, address[] calldata
↪   poolCommitterAddresses)
    external
    override
{
    ...
    for (uint256 i; i < users.length; i++) {
+        require(poolFactory.isValidPoolCommitter(poolCommitterAddresses[i]), "poolCommitter not valid
↪   PoolCommitter");
        ...
    }

    ...
}
```

```
function multiPaidClaimSinglePoolCommitter(address[] calldata users, address poolCommitterAddress)
    external
    override
{

+    require(poolFactory.isValidPoolCommitter(poolCommitterAddress), "poolCommitter not valid
↪   PoolCommitter");
    ...
}
```

```
function withdrawClaimRequest(address poolCommitter) external override {
+    require(poolFactory.isValidPoolCommitter(poolCommitter), "poolCommitter not valid PoolCommitter");
    ....
}
```

Tracer: We are trying to evaluate the benefits of this.

In the case of `multiPaidClaimMultiplePoolCommitters()`,

- More gas: this would be an extra check for every iteration
- `checkClaim()` will return `false`, meaning nothing will happen when the `claim()` function is called
- Correct implementations of the off-chain auto claiming bots will use correct data

The same can be applied to `multiPaidClaimSinglePoolCommitter()`, but only one extra check would be required rather than in every iteration. We do not believe we should be doing on-chain stuff to cater for incorrect off-chain bot implementations. In the case of `withdrawClaimRequest()`, I'd be OK with adding this check.

Spearbit: It is less about catering for incorrect off-chain bots, but more about hardening security. We do not think it is immediately obvious why not checking the validity of the pool committers in these functions is not a security issue. You need to argue that:

- Claim requests can only be made by and stored with valid pool committers as an index, so invalid pool committer rewards will always be zero.
- An attacker-controlled pool committer argument does not lead to re-entrancy and other security issues in these functions.
- Also, not checking it might be fine now, but not anymore with future code updates, and it is easy to forget that the pool committers are not checked for validity in all functions.

Of course, it is up to you to decide if the additional security checks are worth the gas costs. You should at least document that you are deliberately not checking the validity here and state the above mentioned reasoning why it is not a security issue in the current code.

Tracer:

> *Claim requests can only be made by and stored with valid pool committers as an index, so invalid pool committer rewards will always be zero.*

We think this is already enforced by including the `onlyPoolCommitter()` modifier in `makePaidClaimRequest()`, so any subsequent claim attempts will only have claim requests that are populated with non-zero data if it was a pool committer deployed by the factory.

> *An attacker-controlled pool committer argument does not lead to re-entrancy and other security issues in these functions*

We believe this should always hold due to the `checkClaim()` call in the `multiPaidClaimMultiplePoolCommitters()` and `multiPaidClaimSinglePoolCommitter()` functions.

> *Also, not checking it might be fine now, but not anymore with future code updates, and it is easy to forget that the pool committers are not checked for validity in all functions.*

This is definitely a good point and we would be happy to add these checks for this reason.

Addressed in issue 359.

Spearbit: Acknowledged.

5.5 Gas Optimization

5.5.1 Some `SMAOracle` and `AutoClaim` state variables can be declared as `immutable`

Severity: Gas Optimization

Context: SMAOracle.sol#L8-L26, AutoClaim.sol#L18

Description: In the `SMAOracle` contract, the `oracle`, `periods`, `observer`, `scaler` and `updateInterval` state variables are not declared as `immutable`.

In the `AutoClaim` contract, the `poolFactory` state variable is not declared as `immutable`.

Since the mentioned variables are only initialized in the contracts' constructors, they can be declared as `immutable` in order to save gas.

Recommendation: Declare mentioned variables as `immutable`.

```
contract SMAOracle is IOracleWrapper {
    /// Price oracle supplying the spot price of the quote asset
-    address public override oracle;
+    address public immutable override oracle;
     ...

    /// Price observer providing the SMA oracle with historical pricing data
-    address public observer;
+    address public immutable observer;

    /// Number of periods to use in calculating the SMA (`k` in the SMA equation)
-    uint256 public periods;
+    uint256 public immutable periods;
    ...
    /// Duration between price updates
-    uint256 public updateInterval;
+    uint256 public immutable updateInterval;

-    int256 public scaler;
+    int256 public immutable scaler;

    ...
}
```

```
contract AutoClaim is IAutoClaim {
     ...
-   IPoolFactory internal poolFactory;
+   IPoolFactory internal immutable poolFactory;
    ...
}
```

Tracer: We are submitting PR 406 as a mitigation for this. It is a slightly larger PR than we originally intended so as a result it will likely be submitted for several defects here. We would appreciate if each defect could be assessed against it.

Spearbit: Correctly changed to `immutable`, except for `AutoClaim`, which is not part of PR 406.

Tracer: Good point. Addressed in PR 429.

5.5.2 Use of counters can be optimized

Severity: Gas Optimization

Context: PoolCommitter.sol#L504-L522

Description: `counter` and `i` are both used as counters for the same loop.

```
uint32 counter = 1;
uint256 i = 0;
...
while (i < upperBound) {
...
    unchecked {
        counter += 1;
    }
    i++;
}
```

Recommendation: Keep only one counter.

```
- uint32 counter = 1;
+ uint256 counter = 1;
- uint256 i = 0;
...
- while (i < upperBound) {
+ while (counter <= upperBound) {
...
-   i++
}
```

Tracer: This has already been addressed as part of one of the CARE programs. The current state of the function can be seen at PoolCommitter.sol#L481

Spearbit: The double counter issues have been fixed.

Note: `_updateIntervalId` is assigned but is not used in PoolCommitter.sol#L517

Note: The bounded loop has been replaced by an unbounded loop in PoolCommitter.sol#L514, which introduces the risk of the function running out of gas.

5.5.3 `transferOwnership()` function is inaccessible

Severity: Gas Optimization

Context: ERC20_Cloneable.sol#L89-L92

Description: The `ERC20_Cloneable` contract contains a `transferOwnership()` function that may only be called by the owner, which is `PoolFactory`. However `PoolFactory` doesn't call the function so it is essentially dead code, making the deployment cost unnecessary additional gas.

```
function transferOwnership(address _owner) external onlyOwner {
    require(_owner != address(0), "Owner: setting to 0 address");
    owner = _owner;
}
```

Recommendation: Double-check if there is any use for the `transferOwnership()` function. If so, make it accessible, otherwise remove the function.

Tracer: Valid, will fix.

5.5.4 Use cached values when present

Severity: Gas Optimization

Context: PoolCommitter.sol#L609-L632

Description: The `updateAggregateBalance()` function creates a temporary variable `id` with the value `currentIntervalIds[i]`. Immediately after that, `currentIntervalIds[i]` is used again. This could be replaced by `id` to save gas.

```
 function updateAggregateBalance(address user) public override checkInvariantsAfterFunction {
        ...
        for (uint256 i = 0; i < unAggregatedLength; i++) {
            uint256 id = currentIntervalIds[i];
            if (currentIntervalIds[i] == 0) { // could use id
                continue;
            }
```

Recommendation: Code should be changed to:

```
function updateAggregateBalance(address user) public override checkInvariantsAfterFunction {
    ...
            uint256 id = currentIntervalIds[i];
-            if (currentIntervalIds[i] == 0) {
+            if (id == 0) {

}
```

Tracer: Valid, fixed in PR 330.

5.5.5 `_invariantCheckContract` stored twice

Severity: Gas Optimization

Context: PoolCommitter.sol, LeveragedPool.sol

Description: Both the `PoolCommitter` and `LeveragedPool` contracts store the value of `_invariantCheckContract` twice, both in `invariantCheckContract` and `invariantCheck`. This is not necessary and costs extra gas.

```
contract PoolCommitter is IPoolCommitter, Initializable {
    ...
    address public invariantCheckContract;
    IInvariantCheck public invariantCheck;
    ...
    function initialize( ..., address _invariantCheckContract, ... ) external override initializer {
        ...
        invariantCheckContract = _invariantCheckContract;
        invariantCheck = IInvariantCheck(_invariantCheckContract);
        ...
    }
}
```

```
contract LeveragedPool is ILeveragedPool, Initializable, IPausable {
    ...
    address public invariantCheckContract;
    IInvariantCheck public invariantCheck;
    ...
    function initialize(ILeveragedPool.Initialization calldata initialization) external override
↪   initializer {
        ...
        invariantCheckContract = initialization._invariantCheckContract;
        invariantCheck = IInvariantCheck(initialization._invariantCheckContract);
    }
}
```

Recommendation: Store the value of `_invariantCheckContract` once and use typecasts to convert it to the required type.

Tracer: Valid, fixed by PR 398.

5.5.6 Unnecessary `if/else` statement in `LeveragedPool`

Severity: Gas Optimization

Context: LeveragedPool.sol#L352

Description: A boolean variable is used to indicate the type of token to mint. The `if/else` statement can be avoided by using `LONG_INDEX` or `SHORT_INDEX` as the parameter instead of a `bool` to indicate the use of long or short token.

```
uint256 public constant LONG_INDEX = 0;
uint256 public constant SHORT_INDEX = 1;
...
function mintTokens(bool isLongToken,...){
    if (isLongToken) {
        IPoolToken(tokens[LONG_INDEX]).mint(...);
    } else {
        IPoolToken(tokens[SHORT_INDEX]).mint(...);
...
```

Recommendation: The `bool isLongToken` parameter should be replaced by `uint256 tokenType`, and the `if/else` statement should be removed.

Note: `uint8 tokenType` might be even more efficient on Arbitrum.

```
- function mintTokens(bool isLongToken, ...){
+ function mintTokens(uint256 tokenType, ...){
-     if (isLongToken) {
-     IPoolToken(tokens[LONG_INDEX]).mint(...);
+         IPoolToken(tokens[tokenType]).mint(...);
-     } else {
-         IPoolToken(tokens[SHORT_INDEX]).mint(...);
...
```

Tracer: Valid, fixed in PR 338.

5.5.7 Uncached array length used in loop

Severity: Gas Optimization

Context: AutoClaim.sol#L115

Description: The `users` array length is used in a `for` loop condition, therefore the length of the array is evaluated in every loop iteration. Evaluating it once and caching it can save gas.

```
for (uint256 i; i < users.length; i++) { ... }
```

Recommendation: Code should be changed to:

```
- for (uint256 i; i < users.length; i++) {
+ uint256 nrUsers = user.length;
+ for (uint256 i; i < nrUsers; i++) {
```

Tracer: Valid, fixed in PR 331.

5.5.8 Unnecessary deletion of array elements in a loop is expensive

Severity: Gas Optimization

Context: PoolCommitter.sol#L653

Description: The `unAggregatedCommitments[user]` array is deleted after the `for` loop in `updateAggregateBalance`. Therefore, deleting the array elements one by one with `delete unAggregatedCommitments[user][i];` in the loop body costs unnecessary gas.

Recommendation: Array element deletions in the loop should be removed.

```
function updateAggregateBalance(address user) public override checkInvariantsAfterFunction {
    ...

    for (uint256 i = 0; i < unAggregatedLength; i++) {
        ...
        if (commitment.updateIntervalId < updateIntervalId) {
          ...
-         delete unAggregatedCommitments[user][i];
        } else {
          ...
        }
    }

    delete unAggregatedCommitments[user];
    ...
}
```

Tracer: Valid, fixed in PR 385.

5.5.9 Zero-value transfers are allowed

Severity: Gas Optimization

Context: AutoClaim.sol#L77

Description: Given that `claim()` can return `0` when the claim isn't valid yet due to `updateInterval`, the return value should be checked to avoid doing an unnecessary `sendValue()` call with amount `0`.

```
Address.sendValue(
    payable(msg.sender),
    claim(user, poolCommitterAddress, poolCommitter, currentUpdateIntervalId)
);
```

Recommendation: A check should be added to ensure the amount to transfer is greater than `0` before calling `Address.sendValue()`.

Tracer: Valid, fixed in PR 382.

5.5.10 Unneeded `onlyUnpaused` modifier in `setQuoteAndPool()`

Severity: Gas Optimization

Context: PoolCommitter.sol#L776

Description: The `setQuoteAndPool()` function is only callable once, from the factory contract during deployment, due to the `onlyFactory` modifier. During this call, the contract is always unpaused, therefore the `onlyUnpaused` modifier is not necessary.

Recommendation: The `onlyUnpaused` modifier should be removed.

```
- function setQuoteAndPool(address _quoteToken, address _leveragedPool) external override onlyFactory
↪  onlyUnpaused {
+ function setQuoteAndPool(address _quoteToken, address _leveragedPool) external override onlyFactory {
```

Tracer: Valid, will fix.

5.5.11 Unnecessary mapping access in `AutoClaim.makePaidClaimRequest()`

Severity: Gas Optimization

Context: AutoClaim.sol#L46-L62

Description: Resolving mappings consumes more gas than directly accessing the storage `struct`, therefore it's more gas-efficient to use the already de-referenced variable than to resolve the mapping again.

```
function makePaidClaimRequest(address user) external payable override onlyPoolCommitter {
   ClaimRequest storage request = claimRequests[user][msg.sender];
   ...
   uint256 reward = claimRequests[user][msg.sender].reward;
   ...
   claimRequests[user][msg.sender].updateIntervalId = requestUpdateIntervalId;
   claimRequests[user][msg.sender].reward = msg.value;
```

Recommendation: `claimRequests[user][msg.sender]` should be replaced by `request`.

```
- uint256 reward = claimRequests[user][msg.sender].reward;
+ uint256 reward = request.reward;
...
- claimRequests[user][msg.sender].updateIntervalId = requestUpdateIntervalId;
+ request.updateIntervalId = requestUpdateIntervalId;

- claimRequests[user][msg.sender].reward = msg.value;
+ request.reward = msg.value;
```

Tracer: Valid, fixed in PR 397.

5.5.12 Function complexity can be reduced from linear to constant by rewriting loops

Severity: Gas Optimization

Context: PriceObserver.sol#L71-L88, PriceObserver.sol#L130-L146, SMAOracle.sol#L110-L142

Description: The `add()` function of the `PriceObserver` contract shifts an entire array if the buffer is full, and the `SMA()` function of the `SMAOracle` contract sums the values of an array to calculate its average.

Both of these functions have O(n) complexity and could be rewritten to have O(1) complexity. This would save gas and possibly increase the buffer size.

```
contract PriceObserver is Ownable, IPriceObserver {
     ...
     * @dev If the backing array is full (i.e., `length() == capacity()`, then
     *      it is rotated such that the oldest price observation is deleted
    function add(int256 x) external override onlyWriter returns (bool) {
        ...
        if (full()) {
            leftRotateWithPad(x);
        ...
    }
    function leftRotateWithPad(int256 x) private {
        uint256 n = length();
        /* linear scan over the [1, n] subsequence */
        for (uint256 i = 1; i < n; i++) {
            observations[i - 1] = observations[i];
        }
        ...
    }
```

```
contract SMAOracle is IOracleWrapper {
    * @dev O(k) complexity due to linear traversal of the final `k` elements of `xs`
   ...
   function SMA(int256[24] memory xs, uint256 n, uint256 k) public pure returns (int256) {
       ...
       /* linear scan over the [n - k, n] subsequence */
       for (uint256 i = n - k; i < n; i++) {
           S += xs[i];
       }
       ...
   }
}
```

Recommendation: A circular buffer should be used, like other protocols do. This reduces the complexity of the functions to O(1).

Note: This is also suggested in Runtime verification report: B14 PriceObserver - potential gas optimization

Keeping the sum of the last `k` (periods) elements of observations should be considered. In this case, this code should be used to add an element to the array:

```
sum = sum + latest added element - element [buffer length-k]
```

Then the average of the last `k` elements can be calculated by dividing the sum by `k`.

Note: this requires a tighter integration between `SMAOracle` and `PriceObserver`.

Tracer: We are wondering if `PriceObserver` is needed.

The initial motivation was to support potentially multiple "thin" oracles (e.g. `EMAOracle`, etc.) that apply some very simple transformations to the underlying spot price observations. However,

1. It is currently unclear if this is even a product necessity
2. It adds complexity

The question then is **do you consider `PriceObserver`'s existence justified**?

Regardless of the answer to this question, it is pretty clear that `SMAOracle` needs to be refactored to capture asymptotic complexity bounds.

Spearbit: Agreed, the `PriceObserver` is tightly coupled to its `writer`, which is the only contract that can decide *when* it is updated. There is an implicit `updateInterval` and token pair dependency in the `PriceObserver` already which restricts reusing it to "thin oracles" that operate on the same values.

Currently, we would say that the existence of PriceObserver is not justified, and it could just be an abstract contract that SMAOracle inherits. It really depends on how likely it is that there will be other "thin oracle wrappers" in the future and if these will be operating on the same tokens and update intervals. Even in that case, as the observer has only one writer, it will not work well with multiple thin oracles. This is because if two oracles use the same observer, you need to decide which one is the writer and communicate to the other that it shall not write.

You may need multiple writers, but they do not know when the last update to the observer occurred, and you risk writing to it several times before the update interval passed. So we do not think it is reusable in its current form and does not need to exist or be deployed on its own.

Tracer: We are submitting PR 406 as a mitigation for this. It is a slightly larger PR than we originally intended so as a result it will likely be submitted for several defects here. We would appreciate if each defect could be assessed against it.

Spearbit: PriceObserver.sol is removed and an alternative way is found to store prices without shifting.

Note: _calculateSMA() is still O(n).

5.5.13 Unused observer state variable in PoolKeeper

Severity: Gas Optimization

Context: PoolKeeper.sol#L37

Description: There is no use for the observer state variable. It is only used in performUpkeepSinglePool in a require statement to check if is set.

```
address public observer;

function setPriceObserver(address _observer) external onlyOwner {
    ...
    observer = _observer;
    ...

function performUpkeepSinglePool(...)
    require(observer != address(0), "Observer not initialized");
    ...
```

Recommendation: The observer state variable should be removed, along with the setter function and the require statement in performUpkeepSinglePool.

Tracer: Valid, fixed in PR 273.

5.5.14 Usage of temporary variable instead of type casting in PoolKeeper.performUpkeepSinglePool()

Severity: Gas Optimization

Context: PoolKeeper.sol#L132

Description: The pool temporary variable is used to cast the address to ILeveragedPool. Casting the address directly where the pool variable is used saves gas, as _pool is calldata.

Recommendation: A direct typecast from _pool should be used instead of a temporary variable.

```
- ILeveragedPool pool = ILeveragedPool(_pool);
...
- IOracleWrapper poolOracleWrapper = IOracleWrapper(pool.oracleWrapper());
+ IOracleWrapper poolOracleWrapper = IOracleWrapper( ILeveragedPool(_pool).oracleWrapper() );
...
- try pool.poolUpkeep(...)
+ try  ILeveragedPool(_pool).poolUpkeep(...)
```

Tracer: Valid, addressed in issue 365.

5.5.15 Events and event emissions can be optimized

Severity: Gas Optimization

Context: PoolCommitter.sol#L151, PoolCommitter.sol#L776-L783, PoolFactory.sol#L164

Description: Having a single `DeployCommitter` event to be emitted after `setQuoteAndPool()` in `PoolFactory.deployPool()` would result in:

1. Having better UX/event tracking and alignment with the current behavior to emit events during the Factory deployment.
2. Removing the `QuoteAndPoolChanged` event that is emitted only once during the lifetime of the `PoolCommitter` during `PoolFactory.deployPool()`.
3. Removing the `ChangeIntervalSet` emission in `PoolCommitter.initialize()`. The `changeInterval` has not really changed, it was initialized. This can be tracked by the `DeployCommitter` event.

Recommendation: A `DeployCommitter` event emission should be added after the `setQuoteAndPool()` call in `PoolFactory.deployPool()`. That specific event should track the parameters that need to be tracked and the ones previously tracked by `ChangeIntervalSet` and `QuoteAndPoolChanged`.

```
function deployPool(PoolDeployment calldata deploymentParameters) external override returns (address) {
    ...

    // approve the quote token on the pool commiter to finalise linking
    // this also stores the pool address in the commiter
    IPoolCommitter(poolCommitterAddress).setQuoteAndPool(deploymentParameters.quoteToken, _pool);
+    emit DeployCommitter(eploymentParameters.quoteToken, _pool, changeInterval);
    ...
}
```

`QuoteAndPoolChanged` event emission should be removed from `PoolCommitter.setQuoteAndPool()` along with the `_quoteToken` parameter and the check on `_quoteToken`. This last parameter will always be different to `address(0)`, because it was already checked in the `LeveragedPool` initialization.

```
function setQuoteAndPool(address _quoteToken, address _leveragedPool) external override onlyFactory
↪   onlyUnpaused {
-    require(_quoteToken != address(0), "Quote token address cannot be 0 address");
    require(_leveragedPool != address(0), "Leveraged pool address cannot be 0 address");

    leveragedPool = _leveragedPool;
    tokens = ILeveragedPool(leveragedPool).poolTokens();
-    emit QuoteAndPoolChanged(_quoteToken, _leveragedPool);
}
```

The `QuoteAndPoolChanged` event should be removed, as it won't be used considering `setQuoteAndPool()` is only called by the Factory at deploy time. The method name should be changed to `initQuoteAndPool()` to be more precise with the real meaning. NatSpec comments should be updated accordingly.

`ChangeIntervalSet` event emission should be removed from `PoolCommitter.initialize()`.

```
function initialize(
    address _factory,
    address _invariantCheckContract,
    address _autoClaim,
    address _factoryOwner,
    uint256 _mintingFee,
    uint256 _burningFee,
    uint256 _changeInterval
) external override initializer {
    ...
-    emit ChangeIntervalSet(_changeInterval);
    ...
}
```

Tracer: Valid, addressed in PR 395.

5.5.16 Multi-paid claim rewards should be sent only if nonzero

Severity: Gas Optimization

Context: AutoClaim.sol#L122, AutoClaim.sol#L141

Description: In both `multiPaidClaimMultiplePoolCommitters()` and `multiPaidClaimSinglePoolCommitter()`, there could be cases where the reward sent back to the claimer is zero. In these scenarios, the `reward` value should be checked to avoid wasting gas.

Recommendation: The `reward` should be sent back to `msg.sender` only if it is nonzero.

The check is necessary even if the Tracer team adds a check on `msg.value` (PoolCommitter.sol#L312-L314) because claimers could claim nonexistent commits or commits that have already been claimed.

```
function multiPaidClaimMultiplePoolCommitters(address[] calldata users, address[] calldata
↪   poolCommitterAddresses)
    external
    override
{
    ...

-    Address.sendValue(payable(msg.sender), reward);
+    if (reward != 0) {
+        Address.sendValue(payable(msg.sender), reward);
+    }
}

function multiPaidClaimSinglePoolCommitter(address[] calldata users, address poolCommitterAddress)
    external
    override
{
    ...

-    Address.sendValue(payable(msg.sender), reward);
+    if (reward != 0) {
+        Address.sendValue(payable(msg.sender), reward);
+    }
}
```

Tracer: Valid, fixed in PR 382.

5.5.17 Unnecessary quad arithmetic use where integer arithmetic works

Severity: Gas Optimization

Context: PoolSwapLibrary.sol#L331, PoolKeeper.sol#L245-L261

Description: The `ABDKMathQuad` library is used to compute a division which is then truncated with `toUint()`. Semantically this is equivalent to a standard `uint` division, which is more gas efficient. The same library is also unnecessarily used to compute keeper's reward. This can be safely done by using standard `uint` computation.

```
function appropriateUpdateIntervalId(...)
...
uint256 factorDifference = ABDKMathQuad.toUInt(divUInt(frontRunningInterval, updateInterval));
```

```
function keeperReward(...)
        ...
        int256 wadRewardValue = ABDKMathQuad.toInt(
            ABDKMathQuad.add(
                ABDKMathQuad.fromUInt(_keeperGas),
                ABDKMathQuad.div(
                    (
                        ABDKMathQuad.div(
                            (ABDKMathQuad.mul(ABDKMathQuad.fromUInt(_keeperGas), _tipPercent)),
                            ABDKMathQuad.fromUInt(100)
                        )
                    ),
                    FIXED_POINT
                )
            )
        );
        uint256 decimals = IERC20DecimalsWrapper(ILeveragedPool(_pool).quoteToken()).decimals();
        uint256 deWadifiedReward = PoolSwapLibrary.fromWad(uint256(wadRewardValue), decimals);
```

Recommendation: `ABDKMathQuad` should be replaced with standard `uint` computation where possible.

Tracer: Valid, addressed in PR 391.

5.5.18 Custom errors should be used

Severity: Gas Optimization

Context: Contracts

Description: In the latest Solidity versions it is possible to replace the strings used to encode error messages with custom errors, which are more gas efficient.

```
AutoClaim.sol:              require(poolFactory.isValidPoolCommitter(msg.sender), "msg.sender not valid
↪  PoolCommitter");
AutoClaim.sol:              require(_poolFactoryAddress != address(0), "PoolFactory address == 0");
AutoClaim.sol:              require(poolFactory.isValidPoolCommitter(poolCommitterAddress), "Invalid
↪  PoolCommitter");
AutoClaim.sol:              require(users.length == poolCommitterAddresses.length, "Supplied arrays
↪  must be same length");
ChainlinkOracleWrapper.sol: require(_oracle != address(0), "Oracle cannot be 0 address");
ChainlinkOracleWrapper.sol: require(_deployer != address(0), "Deployer cannot be 0 address");
ChainlinkOracleWrapper.sol: require(_decimals <= MAX_DECIMALS, "COA: too many decimals");
ChainlinkOracleWrapper.sol: require(answeredInRound >= roundID, "COA: Stale answer");
ChainlinkOracleWrapper.sol: require(timeStamp != 0, "COA: Round incomplete");
ERC20_Cloneable.sol:        require(msg.sender == owner, "msg.sender not owner");
ERC20_Cloneable.sol:        require(_owner != address(0), "Owner: setting to 0 address");
InvariantCheck.sol:         require(_factory != address(0), "Factory address cannot be null");
InvariantCheck.sol:         require(poolFactory.isValidPool(poolToCheck), "Pool is invalid");
LeveragedPool.sol:          require(!paused, "Pool is paused");
LeveragedPool.sol:           require(!paused, "Pool is paused");
LeveragedPool.sol:           require(!paused, "Pool is paused");
LeveragedPool.sol:           require(msg.sender == keeper, "msg.sender not keeper");
LeveragedPool.sol:           require(msg.sender == invariantCheckContract, "msg.sender not
↪   invariantCheckContract");
LeveragedPool.sol:           require(msg.sender == poolCommitter, "msg.sender not poolCommitter");
LeveragedPool.sol:           require(msg.sender == governance, "msg.sender not governance");
LeveragedPool.sol:           require(initialization._feeAddress != address(0), "Fee address cannot be 0
↪   address");
LeveragedPool.sol:           require(initialization._quoteToken != address(0), "Quote token cannot be 0
↪   address");
LeveragedPool.sol:           require(initialization._oracleWrapper != address(0), "Oracle wrapper cannot
↪   be 0 address");
LeveragedPool.sol:           require(initialization._settlementEthOracle != address(0), "Keeper oracle
↪   cannot be 0 address");
LeveragedPool.sol:           require(initialization._owner != address(0), "Owner cannot be 0 address");
LeveragedPool.sol:           require(initialization._keeper != address(0), "Keeper cannot be 0 address");
LeveragedPool.sol:           require(initialization._longToken != address(0), "Long token cannot be 0
↪   address");
LeveragedPool.sol:           require(initialization._shortToken != address(0), "Short token cannot be 0
↪   address");
LeveragedPool.sol:           require(initialization._poolCommitter != address(0), "PoolCommitter cannot
↪   be 0 address");
LeveragedPool.sol:           require(initialization._invariantCheckContract != address(0),
↪   "InvariantCheck cannot be 0 address");
LeveragedPool.sol:           require(initialization._fee < PoolSwapLibrary.WAD_PRECISION, "Fee >= 100%");
LeveragedPool.sol:           require(initialization._secondaryFeeSplitPercent <= 100, "Secondary fee
↪   split cannot exceed 100%");
LeveragedPool.sol:           require(initialization._updateInterval != 0, "Update interval cannot be 0");
LeveragedPool.sol:           require(intervalPassed(), "Update interval hasn't passed");
LeveragedPool.sol:           require(account != address(0), "Account cannot be 0 address");
LeveragedPool.sol:           require(msg.sender == _oldSecondaryFeeAddress);
LeveragedPool.sol:           require(_keeper != address(0), "Keeper address cannot be 0 address");
LeveragedPool.sol:           require(_governance != governance, "New governance address cannot be same
↪   as old governance address");
LeveragedPool.sol:           require(_governance != address(0), "Governance address cannot be 0
↪   address");
LeveragedPool.sol:           require(governanceTransferInProgress, "No governance change active");
LeveragedPool.sol:           require(msg.sender == _provisionalGovernance, "Not provisional governor");
LeveragedPool.sol:           require(paused, "Pool is live");
PoolCommitter.sol:           require(!paused, "Pool is paused");
PoolCommitter.sol:           require(msg.sender == governance, "msg.sender not governance");
PoolCommitter.sol:           require(!paused, "Pool is paused");
PoolCommitter.sol:           require(!paused, "Pool is paused");
PoolCommitter.sol:           require(!paused, "Pool is paused");
PoolCommitter.sol:           require(msg.sender == invariantCheckContract, "msg.sender not
↪   invariantCheckContract");
PoolCommitter.sol:           require(msg.sender == factory, "Committer: not factory");
PoolCommitter.sol:           require(msg.sender == leveragedPool, "msg.sender not leveragedPool");
PoolCommitter.sol:           require(msg.sender == user || msg.sender == address(autoClaim), "msg.sender
↪   not committer or AutoClaim");
PoolCommitter.sol:           require(_factory != address(0), "Factory address cannot be 0 address");
PoolCommitter.sol:           require(_invariantCheckContract != address(0), "InvariantCheck address
↪   cannot be 0 address");
PoolCommitter.sol:           require(_autoClaim != address(0), "AutoClaim address cannot be null");
PoolCommitter.sol:           require(_mintingFee < PoolSwapLibrary.WAD_PRECISION, "Minting fee >= 100%");
PoolCommitter.sol:           require(_burningFee < PoolSwapLibrary.WAD_PRECISION, "Burning fee >= 100%");
PoolCommitter.sol:           require(userCommit.balanceLongBurnAmount <= balance.longTokens,
↪   "Insufficient pool tokens");
PoolCommitter.sol:           require(userCommit.balanceShortBurnAmount <= balance.shortTokens,
↪   "Insufficient pool tokens");
```

```
PoolCommitter.sol:          require(userCommit.balanceLongBurnMintAmount <= balance.longTokens,
↪  "Insufficient pool tokens");
PoolCommitter.sol:          require(userCommit.balanceShortBurnMintAmount <= balance.shortTokens,
↪  "Insufficient pool tokens");
PoolCommitter.sol:          require(amount > 0, "Amount must not be zero");
PoolCommitter.sol:          require(_quoteToken != address(0), "Quote token address cannot be 0
↪  address");
PoolCommitter.sol:          require(_leveragedPool != address(0), "Leveraged pool address cannot be 0
↪  address");
PoolFactory.sol:            require(_feeReceiver != address(0), "Address cannot be null");
PoolFactory.sol:            require(_poolKeeper != address(0), "PoolKeeper not set");
PoolFactory.sol:            require(autoClaim != address(0), "AutoClaim not set");
PoolFactory.sol:            require(invariantCheck != address(0), "InvariantCheck not set");
PoolFactory.sol:            require(IOracleWrapper(deploymentParameters.oracleWrapper).deployer() ==
↪  msg.sender,"Deployer must be oracle wrapper owner");
PoolFactory.sol:            require(deploymentParameters.leverageAmount >= 1 &&
↪  deploymentParameters.leverageAmount <= maxLeverage,"PoolKeeper: leveraged amount invalid");
PoolFactory.sol:            require(IERC20DecimalsWrapper(deploymentParameters.quoteToken).decimals()
↪  <= MAX_DECIMALS,"Decimal precision too high");
PoolFactory.sol:            require(_poolKeeper != address(0), "address cannot be null");
PoolFactory.sol:            require(_invariantCheck != address(0), "address cannot be null");
PoolFactory.sol:            require(_autoClaim != address(0), "address cannot be null");
PoolFactory.sol:            require(newMaxLeverage > 0, "Maximum leverage must be non-zero");
PoolFactory.sol:            require(_feeReceiver != address(0), "address cannot be null");
PoolFactory.sol:            require(newFeePercent <= 100, "Secondary fee split cannot exceed 100%");
PoolFactory.sol:            require(_fee <= 0.1e18, "Fee cannot be > 10%");
PoolFactory.sol:            require(_mintingFee <= 1e18, "Fee cannot be > 100%");
PoolFactory.sol:            require(_burningFee <= 1e18, "Fee cannot be > 100%");
PoolFactory.sol:            require(_changeInterval <= 1e18, "Change interval cannot be > 100%");
PoolKeeper.sol:             require(msg.sender == address(factory), "Caller not factory");
PoolKeeper.sol:             require(_factory != address(0), "Factory cannot be 0 address");
PoolKeeper.sol:             require(_observer != address(0), "Price observer cannot be 0 address");
PoolKeeper.sol:             require(firstPrice > 0, "First price is non-positive");
PoolKeeper.sol:             require(observer != address(0), "Observer not initialized");
PoolSwapLibrary.sol:        require(timestamp >= lastPriceTimestamp, "timestamp in the past");
PoolSwapLibrary.sol:        require(price != 0, "price == 0");
PoolSwapLibrary.sol:        require(price != 0, "price == 0");
PoolSwapLibrary.sol:        require(price != 0, "price == 0");
PriceObserver.sol:          require(msg.sender == writer, "PO: Permission denied");
PriceObserver.sol:          require(i < length(), "PO: Out of bounds");
PriceObserver.sol:          require(_writer != address(0), "PO: Null address not allowed");
SMAOracle.sol:              require(_spotOracle != address(0) && _observer != address(0) && _deployer
↪  != address(0),"SMA: Null address forbidden");
SMAOracle.sol:              require(_periods > 0 && _periods <= IPriceObserver(_observer).capacity(),
↪  "SMA: Out of bounds");
SMAOracle.sol:              require(_spotDecimals <= MAX_DECIMALS, "SMA: Decimal precision too high");
SMAOracle.sol:              require(_updateInterval != 0, "Update interval cannot be 0");
SMAOracle.sol:              require(block.timestamp >= lastUpdate + updateInterval, "SMA: Too early to
↪  update");
SMAOracle.sol:              require(k > 0 && k <= n && k <= uint256(type(int256).max), "SMA: Out of
↪  bounds");
```

Recommendation: The use of custom error messages should be considered, as explained on the Solidity Language Blog.

Tracer: Solidity currently doesn't support custom errors very well.

The official advice is to convert all `require`s into conditionals manually but we think this severely harms code readability for virtually no gain.

Spearbit: There is no problem leaving the code as it is.

5.6 Informational

5.6.1 Different `updateInterval`s in `SMAOracle` and pools

Severity: Informational

Context: LeveragedPool, SMAOracle

Description: The `updateInterval`s for the pools and the `SMAOracle`s are different.

If the `updateInterval` for `SMAOracle` is larger than the `updateInterval` for `poolUpkeep()`, then the oracle price update could happen directly after the `poolUpkeep()`.

It is possible to perform permissionless calls to `poll()`. In combination with a delayed `poolUpkeep()` an attacker could manipulate the timing of the `SMAOracle` price, because after a call to `poll()` it can't be called again until `updateInterval` has passed.

```
contract LeveragedPool is ILeveragedPool, Initializable, IPausable {
    function initialize(ILeveragedPool.Initialization calldata initialization) external override
↪   initializer {
        ...
        updateInterval = initialization._updateInterval;
        ...
    }
    function poolUpkeep(... ) external override onlyKeeper {
        require(intervalPassed(), "Update interval hasn't passed");
        ...
    }
    function intervalPassed() public view override returns (bool) {
        unchecked {
            return block.timestamp >= lastPriceTimestamp + updateInterval;
        }
    }
```

```
contract SMAOracle is IOracleWrapper {
    constructor(..., uint256 _updateInterval, ... ) {
        updateInterval = _updateInterval;
    }
    function poll() external override returns (int256) {
        require(block.timestamp >= lastUpdate + updateInterval, "SMA: Too early to update");
        return update();
    }
}
```

Recommendation: `SMAOracle`'s `updateInterval` should be shorter than `poolUpkeep()`'s `updateInterval`, but not too short to prevent SMA buffer rotations in-between `poolUpkeep()` calls.

Tracer: We think the safest option is to enforce strict equality between the intervals: semantically they are basically the same thing and we do not see any added value in allowing them to be separate concepts.

Spearbit: Note though that from the pool's perspective the `oracleWrapper` is specified by the deployer and as such does not have control or guarantee over the `SMAOracle`'s `updateInterval` or over its impact on the price calculation.

Tracer: Presumably the deployer enforces this invariant, but oracle risk remains on deployers as per our threat model regardless. With this being said, we think we should try and develop a safer alternative to avoid errors due to negligence.

5.6.2 Tight coupling between `LeveragedPool` and `PoolCommitter`

Severity: Informational

Context: LeveragedPool, PoolCommitter

Description: The `LeveragedPool` and `PoolCommitter` contracts call each other back and forth. This could be optimized to make the code clearer and perhaps save some gas.

Here is an example:

```
contract LeveragedPool is ILeveragedPool, Initializable, IPausable {
   function poolUpkeep(...)  external override onlyKeeper {
        ...
        IPoolCommitter(poolCommitter).executeCommitments(_boundedIntervals, _numberOfIntervals);
        ...
    }
}
contract PoolCommitter is IPoolCommitter, Initializable {
   function executeCommitments(...) external override onlyPool {
        ...
        uint256 lastPriceTimestamp = pool.lastPriceTimestamp();   // call to first contract
        uint256 updateInterval = pool.updateInterval();           // call to first contract
        ...
   }
}
```

Recommendation: If contract `LeveragedPool` calls contract `PoolCommitter` and contract `PoolCommitter` calls back to contract `LeveragedPool` to get some values, it should be considered to send all relevant values from `LeveragedPool` to `PoolCommitter` in the first call.

Tracer: Valid, fixed by multiple commits in PR 393.

Spearbit: Looks good, with some thoughts:

- There are quite a lot of changes, so it is important to have tests that verify that everything still works as expected.
- It might be useful to add the checks for 0 mints again (e.g. `if (longMintAmount > 0)` and `if (shortMintAmount > 0)`).
- The `checkInvariantsBeforeFunction()` modifier of `mintTokens()` is not called, although `poolUpkeep()` calls the `checkInvariantsAfterFunction()` modifier, so that should not be a problem.
- Why is `balancesAndSupplies` used? It was already there in the previous versions. Is this to prevent stack to deep errors?
- It seems like a lot of variables are used, but that can not be simplified further because the old and the new values are needed for most of them.

Tracer:

> *There are quite a lot of changes, so it is important to have tests that verify that everything still works as expected.*

Agreed. We are fairly confident that our existing tests cover the changes, e.g. this test ensures minting is still happening, this one checks for long balance being updated, etc.

> *It might be useful to add the checks for 0 mints again (e.g. `if (longMintAmount > 0)` and `if (shortMintAmount > 0)`).*

Agreed.

> *Why is `balancesAndSupplies` used? It was already there in the previous versions. Is this to prevent stack to deep errors?*

Yes, that is correct. An alternative solution is to have different functions for processing long burn, long mint, etc. but we figured that would add a reasonable amount of complexity in regards to keeping track of the variables that get updated after each of these numbers are calculated.

Spearbit: Acknowledged, checks for 0 mints (e.g. `if (longMintAmount > 0)` and `if (shortMintAmount > 0)`) were added.

5.6.3 Code in `SMA()` is hard to read

Severity: Informational

Context: SMAOracle.sol#L124-L142

Description: The `SMA()` function checks for `k` being smaller or equal to `uint256(type(int256).max)`, a value somewhat difficult to read. Additionally, the number `24` is hardcoded.

Note: This issue was also mentioned in Runtime Verification report: B15 PriceObserver - avoid magic values

```
    function SMA( int256[24] memory xs, uint256 n, uint256 k) public pure returns (int256) {
        ...
        require(k > 0 && k <= n && k <= uint256(type(int256).max), "SMA: Out of bounds");
        ...
        for (uint256 i = n - k; i < n; i++) {
            S += xs[i];
        }
        ...
    }
```

Recommendation: Code should be changed to:

```
-    function SMA( int256[24]            memory xs, uint256 n, uint256 k) public pure returns (int256) {
+    function SMA( int256[MAX_NUM_ELEMS] memory xs, uint256 n, uint256 k) public pure returns (int256) {
        ...
-        require(k > 0 && k <= n && k <= uint256(type(int256).max), "SMA: Out of bounds");
+        require(k > 0 && k <= n && n <= MAX_NUM_ELEMS, "SMA: Out of bounds");
        ...
        for (uint256 i = n - k; i < n; i++) {
            S += xs[i];
        }
        ...
    }
```

Note: `&& n <= MAX_NUM_ELEMS` could also be removed, because array boundary checks will give an error message if `n` is too large.

Tracer: We are submitting PR 406 as a mitigation for this. It is a slightly larger PR than we originally intended so as a result it will likely be submitted for several defects here. We would appreciate if each defect could be assessed against it.

Spearbit: Acknowledged, readability has improved.

5.6.4 Code is chain-dependant due to fixed block time and no support for EIP-1559

Severity: Informational

Context: PoolKeeper

Description: The `PoolKeeper` contract has several hardcoded assumptions about the chain on which it will be deployed. It has no support for EIP-1559 and doesn't use `block.basefee`. On Ethereum Mainnet the blocktime will change to 12 seconds with the ETH2 merge.

The Secureum CARE-X report also has an entire discussion about other chains.

```
contract PoolKeeper is IPoolKeeper, Ownable {
    ...
    uint256 public constant BLOCK_TIME = 13; /* in seconds */
    ...
    /// Captures fixed gas overhead for performing upkeep that's unreachable
    /// by `gasleft()` due to our approach to error handling in that code
    uint256 public constant FIXED_GAS_OVERHEAD = 80195;
    ...
}
```

Recommendation: Code should be as generic as possible to support multiple chains and future changes. At the very least, assumptions made for different chains should be documented.

Tracer: We have decided that we will not fix this for V2 launch, but it is something that we as a dev team hold quite a disdain for, and we plan to generalise and remove assumptions in the next version.

We've added a NatSpec in PR 405.

Spearbit: Acknowledged.

5.6.5 `ABDKQuad`-related constants defined outside `PoolSwapLibrary`

Severity: Informational

Context: PoolCommitter.sol#L24, PoolCommitter.sol#L31

Description: Some `ABDKQuad`-related constants are defined outside of the `PoolSwapLibrary` while others are shadowing the ones defined inside the library. As all `ABDKQuad`-related logic is contained in the library it's less error prone to have any `ABDKQuad`-related definitions in the same file.

The constant `one` is lowercase, while usually constants are uppercase.

```
contract PoolCommitter is IPoolCommitter, Initializable {
    bytes16 public constant one = 0x3fff0000000000000000000000000000;
   ...
    // Set max minting fee to 100%. This is a ABDKQuad representation of 1 * 10 ** 18
    bytes16 public constant MAX_MINTING_FEE = 0x403abc16d674ec800000000000000000;
}
library PoolSwapLibrary {
    /// ABDKMathQuad-formatted representation of the number one
    bytes16 public constant one = 0x3fff0000000000000000000000000000;
}
```

Recommendation: Code should be changed to:

```
contract PoolCommitter is IPoolCommitter, Initializable {
-    bytes16 public constant one = 0x3fff0000000000000000000000000000;
      ...
-    bytes16 public constant MAX_MINTING_FEE = 0x403abc16d674ec800000000000000000;
+    bytes16 public constant MAX_MINTING_FEE = ABDK1E18;
}
library PoolSwapLibrary {
    /// ABDKMathQuad-formatted representation of the number one
-    bytes16 public constant one = 0x3fff0000000000000000000000000000;
+    bytes16 public constant ONE = 0x3fff0000000000000000000000000000;

+    // This is an ABDKQuad representation of 1 * 10 ** 18
+    bytes16 public constant ABDK1E18= 0x403abc16d674ec800000000000000000;
}
```

Tracer: Valid, fixed in PR 329.

5.6.6 Lack of a state to allow withdrawal of tokens

Severity: Informational

Context: LeveragedPool.sol#L516

Description: Immediately after the invariants don't hold and the pool has been paused, Governance can withdraw the collateral (quote). It might be prudent to create a separate state besides `paused`, such that unpause actions can't happen anymore to indicate withdrawal intention.

Note: the comment in `withdrawQuote()` is incorrect. Pool must be paused.

```
    /**
    ...
     * @dev Pool must not be paused    // comment not accurate
    ...
     */
    ...
    function withdrawQuote() external onlyGov {
        require(paused, "Pool is live");
        IERC20 quoteERC = IERC20(quoteToken);
        uint256 balance = quoteERC.balanceOf(address(this));
        IERC20(quoteToken).safeTransfer(msg.sender, balance);
        emit QuoteWithdrawn(msg.sender, balance);
    }
```

Recommendation: The creation of a separate state besides `paused`, to be able to withdraw the quote, should be considered. This new state should have its own requirements for transitioning from the `pause` state.

The comments of `withdrawQuote()` should be updated.

Tracer: We have decided that we are happy with the two states as they are. Since the pool can now only enter a paused state from invariant checking failing, we would prefer to be able to withdraw the settlement tokens without a timelock.

Spearbit: Acknowledged.

5.6.7 Undocumented frontrunning protection

Severity: Informational

Context: PoolFactory.sol#L93-L174, PoolFactory.sol#L183-L203

Description: In the `deployPool()` function of `PoolFactory` contract, the `IOracleWrapper(deploymentParameters.oracleWrapper).deployer() == msg.sender` check protects against frontrunning the deployment transaction of the pool.

This is because the `poolCommitter`, `LeveragedPool` and the pair tokens' instances are deployed at a deterministic address, calculated from the values of `leverageAmount`, `quoteToken` and `oracleWrapper`.

An attacker cannot frontrun the pool deployment because of the different `msg.sender` address, that causes the `deployer()` check to fail. Alternatively, the attacker will have a different `oracleWrapper`, resulting in a different pool. However, this is not obvious to a casual reader.

```
function deployPool(PoolDeployment calldata deploymentParameters) external override returns (address) {
        ...
        require(
            IOracleWrapper(deploymentParameters.oracleWrapper).deployer() == msg.sender,
            "Deployer must be oracle wrapper owner"
        );
        ...
        bytes32 uniquePoolHash = keccak256(
            abi.encode(
                deploymentParameters.leverageAmount,
                deploymentParameters.quoteToken,
                deploymentParameters.oracleWrapper
            )
        );
        PoolCommitter poolCommitter = PoolCommitter(
            Clones.cloneDeterministic(poolCommitterBaseAddress, uniquePoolHash)
        );
        ...
        LeveragedPool pool = LeveragedPool(Clones.cloneDeterministic(poolBaseAddress, uniquePoolHash));
        ...
      }

function deployPairToken(... ) internal returns (address) {
        ...
        bytes32 uniqueTokenHash = keccak256(
            abi.encode(
                deploymentParameters.leverageAmount,
                deploymentParameters.quoteToken,
                deploymentParameters.oracleWrapper,
                direction
            )
        );
        PoolToken pairToken = PoolToken(Clones.cloneDeterministic(pairTokenBaseAddress,
↪   uniqueTokenHash));
      ...
    }
```

Recommendation: A comment should be added to the `deployPool()` function explaining to users how the frontrunning prevention works. Additionally, this will save time to future auditors and developers.

Tracer: Fixed by documenting frontrunning protection in PR 404.

5.6.8 No event exists for users self-claiming commits

Severity: Informational

Context: AutoClaim.sol#L33-49

Description: There is no event emitted when a user self-claims a previous commit for themselves, in contrast to `claim()` which does emit the `PaidRequestExecution` event.

Recommendation: A `PaidRequestExecution` event should be emitted.

```
+    emit PaidRequestExecution(user, msg.sender, request.reward);
```

Tracer: Valid, will address.

5.6.9 Mixups of types and scaling factors

Severity: Informational

Context: PoolSwapLibrary.sol#L4

Description: There are a few findings that are related to mixups of types or scaling factors. The following types and scaling factors are used:

- `uint` (no scaling)
- `uint` (WAD scaling)
- `ABDKMathQuad`
- `ABDKMathQuad` (WAD scaling)

Solidity >0.8.9's user defined value types could be used to prevent mistakes. This will require several typecasts, but they don't add extra gas costs.

Recommendation: The use of user defined value types should be considered.

Tracer: Agreed.

5.6.10 Missing events for `setInvariantCheck()` and `setAutoClaim()` in `PoolFactory`

Severity: Informational

Context: PoolFactory.sol#L224-L227, PoolFactory.sol#L235-L238,

Description: Events should be emitted for access-controlled critical functions, and functions that set protocol parameters or affect the protocol in significant ways.

Recommendation: Events should be emitted in those functions.

Tracer: Valid, fixed in issue 372.

Spearbit: Acknowledged.

5.6.11 Terminology used for tokens and oracles is not clear and consistent across codebase

Severity: Informational

Context: PoolSwapLibrary.sol#L406-L431, PoolKeeper.sol#L276-L281

Description: Different terms are used across the codebase to address the different tokens, leading to some mixups.

Assuming a pair BTC/USDC is being tracked with WETH as collateral, we think the following definitions apply:

- `collateral token` == `quote token` == `settlement token` == WETH
- `pool token` == `long token` + `short token` == long BTC/USDC + short BTC/USDC

As for the oracles:

- `settlementEthOracle` is the oracle for settlement in ETH (WETH/ETH)
- `oracleWrapper` is the oracle for BTC/USDC

Here is an example of a mixup: The comments in `getMint()` and `getBurn()` are different while their result should be similar. It seems the comment on `getBurn()` has reversed settlement and pool tokens.

```
     * @notice Calculates the number of pool tokens to mint, given some settlement token amount and a
↪    price
      ...
     * @return Quantity of pool tokens to mint
     ...
    function getMint(bytes16 price, uint256 amount) public pure returns (uint256) {
        ...
    }
     * @notice Calculate the number of settlement tokens to burn, based on a price and an amount of
↪    pool tokens
        //settlement & pool seem reversed
     ...
     * @return Quantity of pool tokens to burn
    ...
    function getBurn(bytes16 price, uint256 amount) public pure returns (uint256) {
      ...
    }
```

The `settlementTokenPrice` variable in `keeperGas()` is misleading and not clear whether it is *Eth per Settlement* or *Settlement per Eth*.

```
contract PoolKeeper is IPoolKeeper, Ownable {
    function keeperGas(..) public view returns (uint256) {
        int256 settlementTokenPrice =
↪    IOracleWrapper(ILeveragedPool(_pool).settlementEthOracle()).getPrice();
        ...
    }
}
```

Recommendation: Consistent terminology should be used throughout the code, and checks should be made to prevent potential mixups.

`settlementTokenPrice` should be changed to `settlementPerEth`:

```
contract PoolKeeper is IPoolKeeper, Ownable {
    function keeperGas(..) public view returns (uint256) {
-       int256 settlementTokenPrice =
↪   IOracleWrapper(ILeveragedPool(_pool).settlementEthOracle()).getPrice();
+       int256 settlementPerEth     =
↪   IOracleWrapper(ILeveragedPool(_pool).settlementEthOracle()).getPrice();
        ...
    }
}
```

Tracer: Agreed and valid. Will address.

5.6.12 Incorrect NatSpec and comments

Severity: Informational

Context: PoolSwapLibraryL283-L293, LeveragedPool.sol#L511, LeveragedPool.sol#L47

Description: Some NatSpec documentation and comments contain incorrect or unclear information.

In PoolSwapLibraryL283-L293, the NatSpec for the `isBeforeFrontRunningInterval()` function refers to *uncommitment*, which is not longer supported.

```
     * @notice Returns true if the given timestamp is BEFORE the frontRunningInterval starts,
     *         which is allowed for uncommitment.
     function isBeforeFrontRunningInterval(...)
```

In LeveragedPool.sol#L511 the NatSpec for the `withdrawQuote()` function notes that the pool should not be paused while the `require` checks that it is paused.

```
   * @dev Pool must not be paused
   function withdrawQuote() ... {
      require(paused, "Pool is live");
```

In LeveragedPool.sol#L47 the comment is unclear, as it references a singular *update interval* but the mapping points to arrays.

```
    // The most recent update interval in which a user committed
    mapping(address => uint256[]) public unAggregatedCommitments;
```

In PoolToken.sol#L16-L23 both the order and the meaning of the documentation are wrong.

- The `@param` lines order should be switched.
- `@param amount Pool tokens to burn` should be replaced with `@param amount Pool tokens to mint`
- `@param account Account to burn pool tokens to` should be replaced with `@param account Account to mint pool tokens to`

```
/**
 * @notice Mints pool tokens
- * @param amount Pool tokens to burn
- * @param account Account to burn pool tokens to
+ * @param account Account to mint pool tokens to
+ * @param amount Pool tokens to mint
 */
function mint(address account, uint256 amount) external override onlyOwner {
    ...
}
```

In PoolToken.sol#L25-L32 the order of the `@param` lines is reversed.

```
/**
 * @notice Burns pool tokens
- * @param amount Pool tokens to burn
- * @param account Account to burn pool tokens from
+ * @param account Account to burn pool tokens from
+ * @param amount Pool tokens to burn
 */
function burn(address account, uint256 amount) external override onlyOwner {
    ...
}
```

In PoolFactory.sol#L176-L203 the NatSpec `@param` for `poolOwner` is missing. It would also be suggested to change the parameter name from `poolOwner` to `pool`, since the parameter received from `deployPool` is the address of the pool and not the owner of the pool.

```
/**
 * @notice Deploy a contract for pool tokens
+ * @param pool The pool address, owner of the Pool Token
 * @param leverage Amount of leverage for pool
 * @param deploymentParameters Deployment parameters for parent function
 * @param direction Long or short token, L- or S-
 * @return Address of the pool token
 */
function deployPairToken(
-    address poolOwner,
+    address pool,
    string memory leverage,
    PoolDeployment memory deploymentParameters,
    string memory direction
) internal returns (address) {
    ...

-    pairToken.initialize(poolOwner, poolNameAndSymbol, poolNameAndSymbol, settlementDecimals);
+    pairToken.initialize(pool, poolNameAndSymbol, poolNameAndSymbol, settlementDecimals);

    ...
}
```

In PoolSwapLibrary.sol#L433-L454 the comments for two of the parameters of function `getMintWithBurns()` are reversed.

```
     * @param amount ...
     * @param oppositePrice ...
      ...
     function getMintWithBurns(
        ...
        bytes16 oppositePrice,
        uint256 amount,
        ...
    ) public pure returns (uint256) {
     ...
```

In ERC20_Cloneable.sol#L46-L49 a comment at the constructor of contract `ERC20_Cloneable` mentions a default value of `18` for decimals. However, it doesn't use this default value, but the supplied parameter.

Moreover, a comment at the constructor of `ERC20_Cloneable` contract mentions `_setupDecimals`. This is probably a reference to an old version of the OpenZeppelin ERC20 contracts, and no longer relevant.

Additionally, the comments say the values are immutable, but they are set in the `initialize()` function.

```
      @dev Sets the values for {name} and {symbol}, initializes {decimals} with
     * a default value of 18.
     * To select a different value for {decimals}, use {_setupDecimals}.
     *  All three of these values are immutable: they can only be set once during
     * construction.
    ...
    constructor(string memory name_, string memory symbol_, uint8 decimals_) ERC20(name_, symbol_) {
        _decimals = decimals_;
    }

function initialize(address _pool,  string memory name_,  string memory symbol_,  uint8 decimals_)
↪   external initializer {
        owner = _pool;
        _name = name_;
        _symbol = symbol_;
        _decimals = decimals_;
    }
```

Recommendation: NatSpec and comments should be corrected.

Tracer: Valid, addressed in issue 376.

Spearbit: Acknowledged.

6 Additional Comments

7 Appendix