Solidly Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Dacian

Carlitox477

April 20, 2024

Contents

1 About Cyfrin

Cyfrin is a Web3 security company dedicated to bringing industry-leading protection and education to our partners and their projects. Our goal is to create a safe, reliable, and transparent environment for everyone in Web3 and DeFi. Learn more about us at cyfrin.io.

2 Disclaimer

The Cyfrin team makes every effort to find as many vulnerabilities in the code as possible in the given time but holds no responsibility for the findings in this document. A security audit by the team does not endorse the underlying business or product. The audit was time-boxed and the review of the code was solely on the security aspects of the solidity implementation of the contracts.

3 Risk Classification

	Impact: High	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

4 Protocol Summary

Solidly is a UniswapV3-based Concentrated Liquidity Automated Marker Maker (CLAMM) which aims to be more gas-efficient while also offering additional features.

5 Audit Scope

The codebase under audit was split between two components:

- `v3-core` - UniswapV3 fork
- `v3-rewards` - new `RewardsDistributor` contract

Solidly's design involves numerous off-chain components which were not under audit:

- Just-In-Time Liquidity Protection
- Impermanent Loss Mitigation
- ERC20 & ERC721 Voting System
- LP & Voter Bribing

For the Enhanced Reward Incentives only the `RewardsDistributor` on-chain component which allows users to collect their rewards was under audit; the actual rewards calculation algorithm is off-chain and was not under audit.

Centralization risks were also generally outside of the scope for findings as they have been covered in previous audits and a certain amount of centralization is part of Solidly's design; by moving processing off-chain this reduces gas costs for on-chain transactions like swaps and mints. Nevertheless permissioned actors and the risks associated with them will be mentioned in the subsequent section.

The following contracts were included in the scope for this audit:

```
v3-core: commit 075c86e4824529055dc4e26b659248cec0d13e89
contracts/SolidlyV3Factory.sol
contracts/SolidlyV3Pool.sol
contracts/SolidlyV3PoolDeployer.sol
contracts/libraries/Position.sol
contracts/libraries/Status.sol
contracts/libraries/Tick.sol
contracts/libraries/TransferHelper.sol
contracts/libraries/Validation.sol

v3-rewards: commit 6dfb435392ffa64652c8f88c98698756ca80cf28
contracts/RewardDistributor.sol
contracts/libraries/Status.sol
```

6 Executive Summary

Over the course of 9 days, the Cyfrin team conducted an audit on the Solidly smart contracts provided by Solidly Labs. In this period, a total of 21 issues were found.

The findings consist of 3 Medium & 4 Low severity issues with the remainder being informational and gas optimizations. All 3 Mediums were related to the `RewardsDistributor` contract; 2 were griefing attacks and the other was related to reward incentive deposits for Fee-On-Transfer tokens.

From the 4 Low findings 3 were related to the `RewardsDistributor` contract; two were griefing attacks and the third was related to the project's configuration allowing older versions of OpenZeppelin with a relevant security vulnerability. The fourth Low and only finding related to `v3-core` was the possibility for more subtle rug-pulls in pools using malicious tokens due to a gas optimization.

Solidly Modifications To Uniswap V3

Solidly's `v3-core` Uniswap fork is a simplified UniswapV3 with reductions in gas usage due to:

- removed oracle-related observation storage
- simplified fee mechanism; design disincentivizes liquidity being split across different pool tiers
- distribution of fees to users is calculated off-chain (reducing on-chain gas usage) and distributed on-chain via a separate `RewardsDistributor` contract
- additional utility in `mint`, `swap` and `burn` functions allowing slippage protection without periphery contracts

There are 2 new permissioned actors:

- Fee Collector - the `RewardsDistributor` contract who collects fees from the pools
- Fee Setter - off-chain bots used to implement Impermanent Loss Mitigation by dynamically changing fees in response to market volatility

Most importantly Solidly inherits UniswapV3's time-tested security as:

- none of the new permissioned actors have the ability to compromise Liquidity Provider deposited funds
- behavior such as swapping, position management & minting/burning appears to be identical to UniswapV3
- UniswapV3's extensive test suite has been preserved and only modified as needed for the removal of Oracle-related functionality and simplified fee mechanism.

On-Chain Reward Distributor

The on-chain `RewardsDistributor` contract collects fees from pools, allows external actors to deposit reward incentives and set the merkle tree root, and allows users to claim their rewards against the current merkle tree root. It contains 5 permissioned actors:

- Owner - can set other permissioned actors and change the claim delay & max incentive parameters
- Root Admin - can change the merkle tree root
- Root Setter A - can change the 'A' root candidate
- Root Setter B - can change the 'B' root candidate
- Claims Pauser - can pause claiming of rewards and send ether from the contract

Permission-less users are able to:

- pause claiming of rewards by depositing ETH collateral; once deposited the Claims Pauser has total control over the ETH collateral
- set the current merkle root to 'A' root candidate if 'A and 'B' root candidates match

When the current merkle root is updated to allow new rewards to be claimed, all previously unclaimed rewards from past reward periods can still be claimed against the new merkle root; the off-chain merkle root calculation accounts for all previously unclaimed rewards.

`RewardsDistributor` has two hashing mechanisms in the `_generateLeaves` and `getRewardKey` functons; symbolic testing using Halmos was conducted as part of the audit to verify that unique outputs are produced for the possible input sets. Additionally a new comprehensive testing suite which achieves 96% coverage for the `RewardsDistributor` contract was created as part of the audit and provided to the project team.

Summary

Project Name	Solidly
Repository	v3-core
Commit	075c86e48245...
Audit Timeline	Jan 12th - Jan 24th
Methods	Manual Review

Issues Found

Critical Risk	0
High Risk	0
Medium Risk	3
Low Risk	4
Informational	8
Gas Optimizations	6
Total Issues	21

Summary of Findings

[M-1] Attacker can abuse `RewardsDistributor::triggerRoot` to block reward claims and unpause a paused state	Resolved

[M-2] `RewardsDistributor` doesn't correctly handle deposits of fee-on-transfer incentive tokens	Resolved
[M-3] Attacker can corrupt `RewardsDistributor` internal accounting forcing LP token incentive deposits to revert for tokens like `cUSDCv3`	Resolved
[L-1] Use low level `call()` to prevent gas griefing attacks when returned data not required	Resolved
[L-2] Check for valid pool in `RewardsDistributor::depositLPSolidEmissions`, `depositLPTokenIncentive` and `_collectPoolFees`	Acknowledged
[L-3] `SolidlyV3Pool::_mint` and `_swap` don't verify tokens were actually received by the pool	Acknowledged
[L-4] Change `v3-rewards/package.json` to require minimum OpenZeppelin v4.9.2 as prior versions had a security vulnerability in Merkle Multi Proof	Resolved
[I-1] Refactor hard-coded max pool fee into a constant as it is used in multiple places	Acknowledged
[I-2] Prefer explicit function for renouncing ownership and 2-step ownership transfer	Acknowledged
[I-3] `require` and `revert` statements should have descriptive reason strings	Acknowledged
[I-4] Functions not used internally could be marked external	Acknowledged
[I-5] Refactor `zeroRoot` declared in multiple functions into a private constant	Resolved
[I-6] Hard-coded pause collateral fee not appropriate for multi-chain usage	Resolved
[I-7] `RewardsDistributor::_claimSingle` should emit `RewardClaimed` using `amountDelta`	Acknowledged
[I-8] `CollateralWithdrawn` and `CollateralDeposited` events should include relevant amounts	Resolved
[G-1] Cache array length outside of loop	Resolved
[G-2] Don't initialize variables with default value	Resolved
[G-3] Prefer `++x` to `x++`	Acknowledged
[G-4] Cache storage variables in memory when read multiple times without being changed	Acknowledged
[G-5] Use multiple requires instead of a single one with multiple statements is better for gas consumption	Acknowledged
[G-6] Optimize away two memory variables in `RewardsDistributor::generateLeaves`	Acknowledged

7 Findings

7.1 Medium Risk

7.1.1 Attacker can abuse `RewardsDistributor::triggerRoot` to block reward claims and unpause a paused state

Description: Consider the code of `RewardsDistributor::triggerRoot`:

```
    function triggerRoot() external {
        bytes32 rootCandidateAValue = rootCandidateA.value;
        if (rootCandidateAValue != rootCandidateB.value || rootCandidateAValue == bytes32(0)) revert
        ↪  RootCandidatesInvalid();
        root = Root({value: rootCandidateAValue, lastUpdatedAt: block.timestamp});
        emit RootChanged(msg.sender, rootCandidateAValue);
    }
```

This function:

- can be called by anyone
- if it succeeds, sets `root.value` to `rootCandidateA.value` and `root.lastUpdatedAt` to `block.timestamp`
- doesn't reset `rootCandidateA` or `rootCandidateB`, so it can be called over and over again to continually update `root.lastUpdatedAt` or to set `root.value` to `rootCandidateA.value`.

Impact: An attacker can abuse this function in 2 ways:

- by calling it repeatedly an attacker can continually increase `root.lastUpdatedAt` to trigger the claim delay revert in `RewardsDistributor::claimAll` effectively blocking reward claims
- by calling it after reward claims have been paused, an attacker can effectively unpause the paused state since `root.value` is over-written with the valid value from `rootCandidateA.value` and claim pausing works by setting `root.value == zeroRoot`.

Recommended Mitigation: Two possible options:

- Make `RewardsDistributor::triggerRoot` a permissioned function such that an attacker can't call it
- Change `RewardsDistributor::triggerRoot` to reset `rootCandidateA.value = zeroRoot` such that it can't be successfully called repeatedly.

Solidly: Fixed in commits 653c196 & 1170eac.

Cyfrin: Verified. One consequence of the updated implementation is that the contract will start in the "paused" state and root candidates will be unable to be set. This means that the admin will have to set the first valid root via `setRoot` in order to "unpause" from the initial state post-deployment.

7.1.2 `RewardsDistributor` doesn't correctly handle deposits of fee-on-transfer incentive tokens

Description: `the kenneth` stated in telegram that Fee-On-Transfer tokens are fine to use as incentive tokens with `RewardsDistributor`, however when receiving Fee-On-Transfer tokens and storing the reward amount the accounting does not account for the fee deducted from the transfer amount in-transit, for example:

```
function _depositLPIncentive(
    StoredReward memory reward,
    uint256 amount,
    uint256 periodReceived
) private {
    IERC20(reward.token).safeTransferFrom(
        msg.sender,
        address(this),
        amount
    );

    // @audit stored `amount` here will be incorrect since it doesn't account for
    // the actual amount received after the transfer fee was deducted in-transit
    _storeReward(periodReceived, reward, amount);
}
```

Impact: The actual reward calculation is done off-chain and is outside the audit scope nor do we have visibility of that code. But events emitted by `RewardsDistributor` and the stored incentive token deposits in `RewardsDistributor::periodRewards` use incorrect amounts for Fee-On-Transfer incentive token deposits.

Recommended Mitigation: In `RewardsDistributor::_depositLPIncentive` & `depositVoteIncentive`:

- read the `before` transfer token balance of `RewardsDistributor` contract
- perform the token transfer
- read the `after` transfer token balance of `RewardsDistributor` contract
- calculate the difference between the `after` and `before` balances to get the true amount that was received by the `RewardsDistributor` contract accounting for the fee that was deducted in-transit
- use the true received amount to generate events and write the received incentive token amounts to `RewardsDistributor::periodRewards`.

Also note that `RewardsDistributor::periodRewards` is never read in the contract, only written to. If it is not used by off-chain processing then consider removing it.

Solidly: Fixed in commit be54da1.

Cyfrin: Verified.

7.1.3 Attacker can corrupt `RewardsDistributor` internal accounting forcing LP token incentive deposits to revert for tokens like `cUSDCv3`

Description: Some tokens like cUSDCv3 contains a special case for `amount == type(uint256).max` in their transfer functions that results in only the user's balance being transferred.

For such tokens in this case incentive deposits via `depositLPTokenIncentive` will transfer less tokens than expected. The consequence of this is if a protocol like Compound wanted to incentivize a pool with a token like `cUSDCv3`, an attacker can front-run their transaction to corrupt the internal accounting forcing it to revert.

Impact: Corrupted accounting for incentive reward deposits with tokens like `cUSDCv3` can be exploited to deny future incentive reward deposits using the same token.

POC: Consider the following functions:

```
function _validateIncentive(
    address token,
    uint256 amount,
    uint256 distributionStart,
    uint256 numDistributionPeriods
) private view {
    // distribution must start on future epoch flip and last for [1, max] periods
    if (
        numDistributionPeriods == 0                  || // Distribution in 0 periods is invalid
        numDistributionPeriods > maxIncentivePeriods || // Distribution over max period is invalid
        distributionStart % EPOCH_DURATION != 0      || // Distribution must start at the beginning of
        ↪  a week
        distributionStart < block.timestamp             // Distribution must start in the future
    ) revert InvalidIncentiveDistributionPeriod();

    // approvedIncentiveAmounts indicates the min amount of
    // tokens to distribute per period for a whitelisted token
    uint256 minAmount = approvedIncentiveAmounts[token] * numDistributionPeriods;

    // @audit validation passes for `amount == type(uint256).max`
    if (minAmount == 0 || amount < minAmount)
        revert InvalidIncentiveAmount();
}

function _depositLPIncentive(
    StoredReward memory reward,
    uint256 amount,
    uint256 periodReceived
) private {
    // @audit does not guarantee that `amount`
    // is transferred if `amount == type(uint256).max`
    IERC20(reward.token).safeTransferFrom(msg.sender, address(this), amount);

    // @audit incorrect `amount` will be stored in this case
    _storeReward(periodReceived, reward, amount);
}
```

If a protocol like Compound wanted to incentivize a pool with a token like `cUSDCv3` for 2 periods:

1. Bob see this in the mempool and calls `RewardsDistributor.depositLPTokenIncentive(pool that compound want to incentivize, cUSDCv3, type(uint256).max, distribution start to DOS, valid numDistributionPeriods)`
2. When Compound try to do a valid call, `_storeReward` will revert because `periodRewards[period][rewardKey] += amount` will overflow since its amount value is `type(uint256).max` due to Bob's front-run transaction.

Recommended mitigation: One possible solution:

1) Divide `_validateIncentive` into 2 functions:

```
function _validateDistributionPeriod(
    uint256 distributionStart,
    uint256 numDistributionPeriods
) private view {
    // distribution must start on future epoch flip and last for [1, max] periods
    if (
        numDistributionPeriods == 0                  || // Distribution in 0 periods is invalid
        numDistributionPeriods > maxIncentivePeriods || // Distribution over max period is invalid
        distributionStart % EPOCH_DURATION != 0      || // Distribution must start at the beginning of
        ↪  a week
        distributionStart < block.timestamp             // Distribution must start in the future
    ) revert InvalidIncentiveDistributionPeriod();
}

// Before calling this function, _validateDistributionPeriod must be called
function _validateIncentive(
    address token,
    uint256 amount,
    uint256 numDistributionPeriods
) private view {
    uint256 minAmount = approvedIncentiveAmounts[token] * numDistributionPeriods;

    if (minAmount == 0 || amount < minAmount)
        revert InvalidIncentiveAmount();
}
```

2) Change `_depositLPIncentive` to return the actual amount received and call `_validateIncentive`:

```
function _depositLPIncentive(
    StoredReward memory reward,
+   uint256 numDistributionPeriods
    uint256 amount,
    uint256 periodReceived
-) private {
+) private returns(uint256 actualDeposited) {
+   uint256 tokenBalanceBeforeTransfer = IERC20(reward.token).balanceOf(address(this));
    IERC20(reward.token).safeTransferFrom(
        msg.sender,
        address(this),
        amount
    );
-   _storeReward(periodReceived, reward, amount);
+   actualDeposited = IERC20(reward.token).balanceOf(address(this)) - tokenBalanceBeforeTransfer;
+   _validateIncentive(reward.token, actualDeposited, numDistributionPeriods);
+   _storeReward(periodReceived, reward, actualDeposited);
}
```

3) Change `depositLPTokenIncentive` to use the new functions, read the actual amount returned and use that in the event emission:

```
function depositLPTokenIncentive(
    address pool,
    address token,
    uint256 amount,
    uint256 distributionStart,
    uint256 numDistributionPeriods
) external {
-   _validateIncentive(
-       token,
-       amount,
-       distributionStart,
-       numDistributionPeriods
-   );
+   // Verify that number of period is and start time is valid
+   _validateDistributionPeriod(
+       uint256 distributionStart,
+       uint256 numDistributionPeriods
+   );
    StoredReward memory reward = StoredReward({
        _type: StoredRewardType.LP_TOKEN_INCENTIVE,
        pool: pool,
        token: token
    });
    uint256 periodReceived = _syncActivePeriod();
-   _depositLPIncentive(reward, amount, periodReceived);
+   uint256 actualDeposited = _depositLPIncentive(reward, amount, periodReceived);

    emit LPTokenIncentiveDeposited(
        msg.sender,
        pool,
        token,
-       amount,
+       actualDeposited
        periodReceived,
        distributionStart,
        distributionStart + (EPOCH_DURATION * numDistributionPeriods)
    );
}
```

This mitigation also resolves the issue related to incorrect accounting for fee-on-transfer tokens.

Solidly: Fixed in commit be54da1.

Cyfrin: Verified.

7.2 Low Risk

7.2.1 Use low level `call()` to prevent gas griefing attacks when returned data not required

Description: Using `call()` when the returned data is not required unnecessarily exposes to gas griefing attacks from huge returned data payload. For example:

```
(bool sent, ) = _to.call{value: _amount}("");
require(sent);
```

Is the same as writing:

```
(bool sent, bytes memory data) = _to.call{value: _amount}("");
require(sent);
```

In both cases the returned data will be copied into memory exposing the contract to gas griefing attacks, even though the returned data is not used at all.

Impact: Contract unnecessarily exposed to gas griefing attacks.

Recommended Mitigation: Use a low-level call when the returned data is not required, eg:

```
bool sent;
assembly {
    sent := call(gas(), _to, _amount, 0, 0, 0, 0)
}
if (!sent) revert FailedToSendEther();
```

Solidly: Fixed in commit be54da1.

Cyfrin: Verified.

7.2.2 Check for valid pool in `RewardsDistributor::depositLPSolidEmissions`, `depositLPTokenIncentive` and `_collectPoolFees`

Description: `RewardsDistributor::depositLPSolidEmissions` and `depositLPTokenIncentive` contain no validation that `pool` is a valid pool address, while `depositVoteIncentive` does perform some validation of the pool parameter. Consider adding validation to ensure LP emissions/incentives are recorded against a valid `pool` parameter.

Similarly `RewardsDistributor::_collectPoolFees` never validates if the pool is legitimate and anyone can call its parent function `collectPoolFees`. An attacker could create their own fake pool which implements `ISolidlyV3PoolMinimal::collectProtocol` but doesn't transfer any tokens just returns large output amounts, and for `token0` and `token1` return the address of popular high-profile tokens.

This could make it appear like `RewardsDistributor` has received significantly more rewards than it actually has by corrupting the event log and `periodRewards` storage location with false information. Consider validating the pool in `RewardsDistributor::_collectPoolFees` and potentially whether `RewardsDistributor` has actually received the tokens.

Also note that `RewardsDistributor::periodRewards` is never read in the contract, only written to. If it is not used by off-chain processing then consider removing it.

Solidly: Acknowledged. The off-chain processor only computes pools that are validated through the factory.

7.2.3 `SolidlyV3Pool::_mint` and `_swap` don't verify tokens were actually received by the pool

Description: Some versions of `SolidlyV3Pool::_mint` & `_swap` don't verify tokens were actually received by the pool. In contrast UniswapV3's equivalent `mint` & `swap` functions always verify tokens were received by the pool.

Impact: Solidly will be more vulnerable to malicious tokens or tokens with non-standard behavior. One possible attack path is a token which has a blacklist that doesn't process transfers for blacklisted accounts but also doesn't revert and simply returns `true`. The token owner can execute a more subtle rug-pull by:

- allowing the pool to grow to a sufficient size
- adding themselves to the blacklist
- calling `swap` to drain the other token without actually transferring any of the malicious token, draining the liquidity pool.

Recommended Mitigation: `_mint` and `_swap` functions should check that the expected token amounts were transferred into the pool.

Solidly: We ommited this on purpose for gas savings since we don't support exotic ERC20s on v3-core. Users can create such a pool if they want since it's permission-less, but it's something we explicitly and officially don't support.

7.2.4 Change `v3-rewards/package.json` to require minimum OpenZeppelin v4.9.2 as prior versions had a security vulnerability in Merkle Multi Proof

Description: `v3-rewards/package.json` currently specifies a minimum OpenZeppelin version of 4.5.0. However some older OZ versions contained a security vulnerability in the Merkle Multi Proof which was fixed in 4.9.2.

Recommended Mitigation: Change `v3-rewards/package.json` to require minimum OpenZeppelin v4.9.2:

```
"@openzeppelin/contracts": "^4.9.2",
```

Solidly: Fixed in commit 6481747.

Cyfrin: Verified.

7.3 Informational

7.3.1 Refactor hard-coded max pool fee into a constant as it is used in multiple places

Description: `100000` is the hard-coded max pool fee. There are two require statements enforcing this hard-coded value in `SolidlyV3Factory::enableFeeAmount` L90 and `SolidlyV3Pool::setFee` L794.

Using the same hard-coded value in multiple places throughout the code is error-prone as when making future code updates a developer can easily update one place but forget to update the others; recommend refactoring to use a constant which can be referenced instead of hard-coding.

Solidly: Acknowledged.

7.3.2 Prefer explicit function for renouncing ownership and 2-step ownership transfer

Description: `RewardsDistributor::setOwner` and `SolidlyV3Factory::setOwner` allow the current owner to brick the ownership by setting `owner = address(0)`, which would prevent future access to admin functionality. Prefer an explicit function for renouncing ownership to prevent this occurring by mistake and prefer a 2-step ownership transfer mechanism. Both of these features are available in OZ Ownable2Step.

Solidly: Acknowledged.

7.3.3 `require` and `revert` statements should have descriptive reason strings

Description: `require` and `revert` statements should have descriptive reason strings:

```
File: SolidlyV3Factory.sol

46:         require(tokenA != tokenB);

48:         require(token0 != address(0));

50:         require(tickSpacing != 0);

51:         require(getPool[token0][token1][tickSpacing] == address(0));

61:         require(msg.sender == owner);

68:         require(msg.sender == owner);

75:         require(msg.sender == owner);

88:         require(msg.sender == owner);

89:         require(fee <= 100000);

94:         require(tickSpacing > 0 && tickSpacing < 16384);

95:         require(feeAmountTickSpacing[fee] == 0);
```

```
File: SolidlyV3Pool.sol

116:         require(success && data.length >= 32);

127:         require(success && data.length >= 32);

302:         require(amount > 0);

327:         require(amount > 0);

947:         require(fee <= 100000);
```

```
File: libraries/FullMath.sol

34:             require(denominator > 0);

43:         require(denominator > prod1);

120:             require(result < type(uint256).max);
```

```
File: RewardsDistributor.sol

564:         require(sent);

595:         require(success && data.length >= 32);
```

Solidly: Acknowledged.

7.3.4 Functions not used internally could be marked external

Description: Functions not used internally could be marked external:

```
File: SolidlyV3Factory.sol

87:     function enableFeeAmount(uint24 fee, int24 tickSpacing) public override {
```

Solidly: Acknowledged.

7.3.5 Refactor `zeroRoot` declared in multiple functions into a private constant

Description: `zeroRoot` is declared and used in `RewardsDistributor::pauseClaimsGovernance` L546 and `pauseClaimsPublic` L554. Consider refactoring it into a private constant to avoid declaring it in multiple functions.

Solidly: Fixed in commit 653c196.

Cyfrin: Verified.

7.3.6 Hard-coded pause collateral fee not appropriate for multi-chain usage

Description: As Solidly aims to be multi-chain in the future, hard-coding a pause collateral fee of 5 ether in `RewardsDistributor::pauseClaimsPublic` may not be appropriate on other chains as this amount would represent

very little value. Consider having a `public` storage variable for the pause collateral fee and an `onlyOwner` function to set it.

Solidly: Fixed in commit 653c196.

Cyfrin: Verified.

7.3.7 `RewardsDistributor::_claimSingle` should emit `RewardClaimed` using `amountDelta`

Description: In `RewardsDistributor::_claimSingle`, the `amount` parameter gets subtracted from the `previouslyClaimed` parameter. Consider the case where a user is entitled to 10 reward tokens.

The user claims their 10 tokens.

Then later on the user becomes entitled to another 10 tokens for the same pool/token/type (`rewardKey`). If the user tries to claim with `amount = 10` this would now fail; the user must claim with `amount = 20` to pass the subtraction of the previously claimed amount.

This design seems kind of confusing; users have to keep track of the total amount they have claimed, then add to that the new amount they can claim, and call claim with that total amount.

Even though only the difference `amountDelta` is sent to the user, the `RewardClaimed` event is emitted with `amount`. So in the above scenario there would be two `RewardClaimed` events emitted with `amount (10)` and `amount(20)` even though the user only received 20 total reward tokens.

Consider refactoring this function such that users can simply call it with the amount they are entitled to claim, or at least changing the event emission to use `amountDelta` instead of `amount`.

Solidly: Acknowledged.

7.3.8 `CollateralWithdrawn` and `CollateralDeposited` events should include relevant amounts

Description: In `RewardsDistributor::withdrawCollateral` add the `_amount` parameter when emitting the `CollateralWithdrawn` event. This is required as the amount sent does not have to be the same as the amount deposited.

Consider adding the amount deposited to the `CollateralDeposited` event as well, since the required collateral amount could be changed meaning that the current value may not be true for every collateral deposit that has occurred.

Solidly: Fixed in commit 6481747.

Cyfrin: Verified.

7.4 Gas Optimization

7.4.1 Cache array length outside of loop

Description: Cache array length outside of loop:

```
File: contracts/RewardsDistributor.sol
// @audit use `numLeaves` from L263 instead of `earners.length`
265:         for (uint256 i; i < earners.length; ) {
```

Solidly: Fixed in commit 6481747.

Cyfrin: Verified.

7.4.2 Don't initialize variables with default value

Description: Don't initialize variables with default value:

```
File: contracts/RewardsDistributor.sol

184:         for (uint256 i = 0; i < numClaims; ) {
```

```
File: libraries/TickMath.sol

67:         uint256 msb = 0;
```

Solidly: Fixed in commit 6481747 for `RewardsDistributor`; v3-core is already deployed and not upgradeable.

Cyfrin: Verified.

7.4.3 Prefer `++x` to `x++`

Description: Prefer `++x` to `x++`:

File: `TickBitmap.sol`

```
48:         if (tick < 0 && tick % tickSpacing != 0) compressed--; // round towards negative infinity
```

File: `SolidlyV3Pool.sol`

```
965:             if (amount0 == poolFees.token0) amount0--; // ensure that the slot is not cleared, for
↪  gas savings
```

File: `SolidlyV3Pool.sol`

```
970:             if (amount1 == poolFees.token1) amount1--; // ensure that the slot is not cleared, for
↪  gas savings
```

File: `FullMath.sol`

```
120:             result++;
```

Solidly: Acknowledged.

7.4.4 Cache storage variables in memory when read multiple times without being changed

Description: Cache storage variables in memory when read multiple times without being changed:

File: `SolidlyV3Pool.sol`

```
// @audit no need to load `slot0.fee` twice from storage since it doesn't change;
// load from storage once into memory then use in-memory copy
913:        uint256 fee0 = FullMath.mulDivRoundingUp(amount0, slot0.fee, 1e6);
914:        uint256 fee1 = FullMath.mulDivRoundingUp(amount1, slot0.fee, 1e6);


// @audit `poolFees.token0` and `poolFees.token1` are read from storage multiple times
// but don't get changed until L966 & L971. Load them both from storage once into memory
// then use the in-memory copy instead of repeatedly reading the same value from storage
961:        amount0 = amount0Requested > poolFees.token0 ? poolFees.token0 : amount0Requested;
962:        amount1 = amount1Requested > poolFees.token1 ? poolFees.token1 : amount1Requested;

964:        if (amount0 > 0) {
965:            if (amount0 == poolFees.token0) amount0--; // ensure that the slot is not cleared, for
↪  gas savings
966:            poolFees.token0 -= amount0;
967:            TransferHelper.safeTransfer(token0, recipient, amount0);
968:        }
969:        if (amount1 > 0) {
970:            if (amount1 == poolFees.token1) amount1--; // ensure that the slot is not cleared, for
↪  gas savings
971:            poolFees.token1 -= amount1;
972:            TransferHelper.safeTransfer(token1, recipient, amount1);
973:        }
```

File: `SolidlyV3Factory.sol`

```
// @audit `owner` is read from storage twice returning the same value each time. Read it from
// storage once into memory, then use the in-memory copy both times
61:        require(msg.sender == owner);
62:        emit OwnerChanged(owner, _owner);
```

Solidly: Acknowledged.

7.4.5 Use multiple requires instead of a single one with multiple statements is better for gas consumption

Description: Use multiple requires instead of a single one with multiple `&&` is better for gas consumption. The reason is because `require` is translated as revert which does not consume gas if it reverts. However `&&` consume gas. Therefore, opting for multiple require is more gas efficient than opting for a single one with multiple statements that mus be true.

```
// SolidlyV3Pool.sol
116:    require(success && data.length >= 32);
127:    require(success && data.length >= 32);
278:    require(amount0 >= amount0Min && amount1 >= amount1Min, 'AL');
293:    require(amount0 >= amount0Min && amount1 >= amount1Min, 'AL');
391:     require(amount0FromBurn >= amount0FromBurnMin && amount1FromBurn >= amount1FromBurnMin, 'AL');
456:    require(amount0 >= amount0Min && amount1 >= amount1Min, 'AL');

// RewardsDistributor.sol
607:    require(success && data.length >= 32);
```

Solidly: Acknowledged.

7.4.6 Optimize away two memory variables in `RewardsDistributor::generateLeaves`

Description: Optimize away two memory variables in `RewardsDistributor::generateLeaves` by using a named return variable and removing the temporary `leaf` variable:

```
function _generateLeaves(
    address[] calldata earners,
    EarnedRewardType[] calldata types,
    address[] calldata pools,
    address[] calldata tokens,
    uint256[] calldata amounts
) private pure returns (bytes32[] memory leaves) {
    uint256 numLeaves = earners.length;
    // @audit using named return variable
    leaves            = new bytes32[](numLeaves);

    // @audit using cached array length in loop
    for (uint256 i; i < numLeaves; ) {
        // @audit assign straight to return variable
        leaves[i] = keccak256(
            bytes.concat(
                keccak256(
                    abi.encode(
                        earners[i],
                        types[i],
                        pools[i],
                        tokens[i],
                        amounts[i]
                    )
                )
            )
        );
        unchecked {
            ++i;
        }
    }
    return leaves;
}
```

Solidly: Acknowledged.