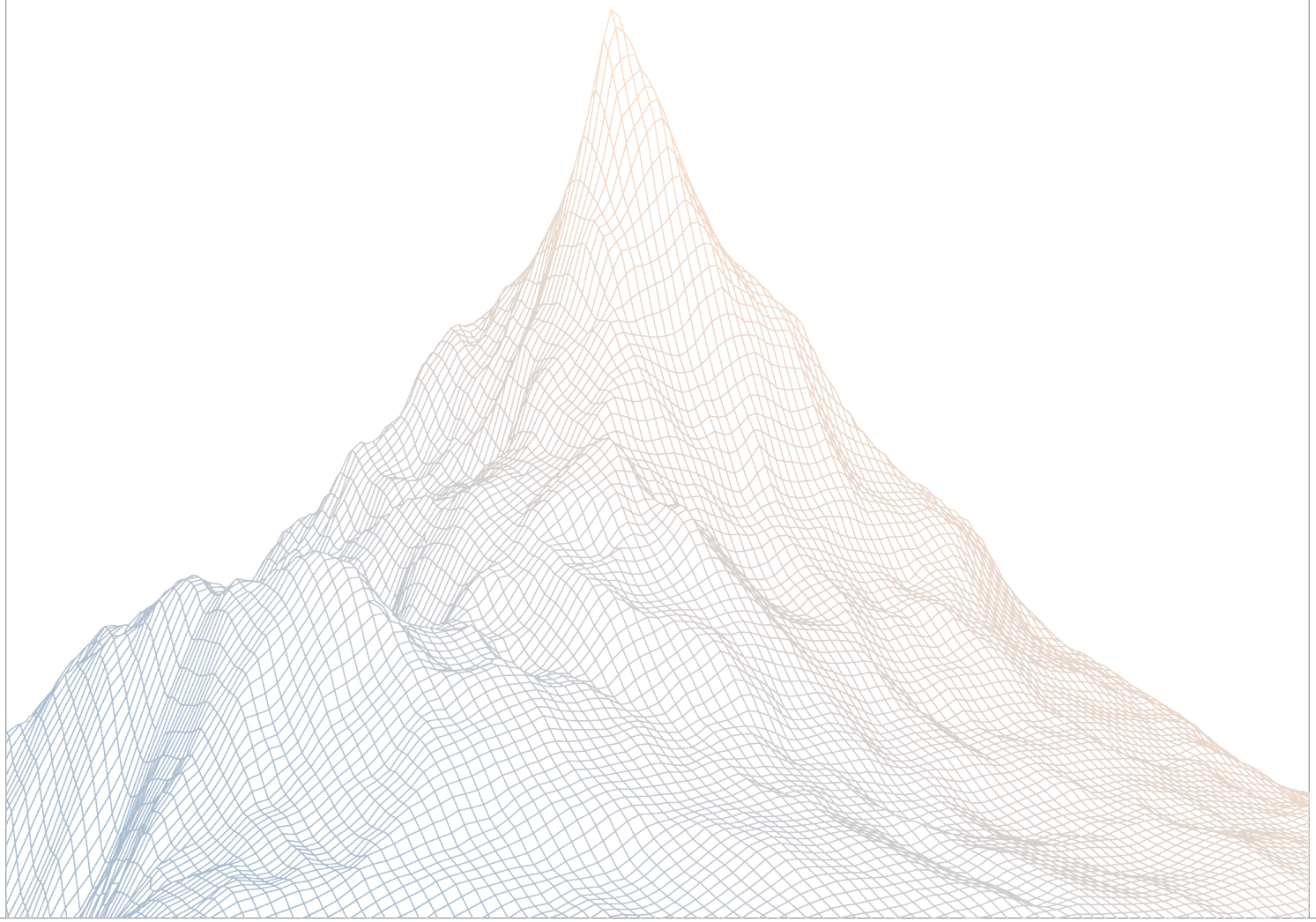
Zenith

Lumiterra

Smart Contract Security Assessment

VERSION 1.1



AUDIT DATES: October 17th to October 26th, 2025

AUDITED BY: 0xAlix2
rvierdiiev

Contents

1 Introduction

1.1 About Zenith

Zenith assembles auditors with proven track records: finding critical vulnerabilities in public audit competitions.

Our audits are carried out by a curated team of the industry's top-performing security researchers, selected for your specific codebase, security needs, and budget.

Learn more about us at https://zenith.security.

1.2 Disclaimer

This report reflects an analysis conducted within a defined scope and time frame, based on provided materials and documentation. It does not encompass all possible vulnerabilities and should not be considered exhaustive.

The review and accompanying report are presented on an "as-is" and "as-available" basis, without any express or implied warranties.

Furthermore, this report neither endorses any specific project or team nor assures the complete security of the project.

1.3 Risk Classification

SEVERITY LEVEL	IMPACT: HIGH	IMPACT: MEDIUM	IMPACT: LOW
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

2

Executive Summary

2.1 About Lumiterra

Lumiterra is a multiplayer, open-world survival crafting game where you can battle, farm with your friends or mysterious collected creatures in a vast world!

2.2 Scope

The engagement involved a review of the following targets:

Target	neuronest-solidity
Repository	https://github.com/LumiterraGame/neuronest-solidity.git
Commit Hash	`d451be958fee3e5273074586c8a6f3ebac6a8e94`
Files	`src/**/*.sol`

2.3 Audit Timeline

October 17, 2025	Audit start
October 26, 2025	Audit end
October 28, 2025	Report published

2.4 Issues Found

SEVERITY	COUNT
Critical Risk	0
High Risk	3
Medium Risk	4
Low Risk	6
Informational	6
Total Issues	**19**

3

Findings Summary

ID	Description	Status
H-1	buyHashCore signature check is switched off allowing anyone to buy a hascore	Resolved
H-2	Survivor rewards distributed in current epoch become unclaimable for early vesting users	Resolved
H-3	lastUserVestingCreatedEpoch may be overwritten skipping vestings creation for previous epochs	Resolved
M-1	Incorrect reward rollback logic, leading to inaccurate survivor reward distribution	Resolved
M-2	Dividend share can be diluted by capped (inactive) keys during fee injections	Resolved
M-3	No ability to change feeManager	Resolved
M-4	Lack of slippage protection forces users to overpay for Hashcore purchases	Resolved
L-1	Outdated epoch data returned by getters when coreRushEpoch end time has passed	Resolved
L-2	LumiTownExtension allows keeper to provide totalPointShare not equal to 100%	Resolved
L-3	Missing EIP-712 version in domain separator allows cross-upgrade Signature	Resolved
L-4	Epoch cannot advance or jackpot be claimed at exact endTime boundary	Resolved
L-5	Vesting entries are never pruned after claim	Resolved
L-6	Missing IERC165 support in supportsInterface()	Resolved
I-1	globalStakeData.globalTotalStakeAmount mixes different agent tokens	Resolved

ID	Description	Status
I-2	placeLiquidity returns a single positionId even when minting multiple LP positions	Resolved
I-3	Changing refreshPeriod without resetting schedule skews next refresh time	Resolved
I-4	Unnecessary payable on deployToken	Resolved
I-5	Unnecessary < 0 Check in buyHashCore Function	Resolved
I-6	Remove unused parameter	Resolved

4

Findings

4.1 High Risk

A total of 3 high risk findings were identified.

[H-1] buyHashCore signature check is switched off allowing anyone to buy a hascore

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- LumiTownExtension.sol

Description:

The `buyHashCore()` function is intended to restrict access to only callers with valid signatures approved by `townConfig.signerAddress`. However, the actual signature verification logic is **commented out**, meaning the function currently does **not** enforce any real authentication:

```
(bool success, bytes32 digest) = _verifySignature(buyHashCoreSignature,
    signature);
// if (!success) revert Invalid();
```

As a result, **anyone** can supply an arbitrary signature and successfully call `buyHashCore()`, bypassing the intended authorization mechanism and purchasing a HashCore without proper validation.

Recommendations:

Uncomment and enforce the signature verification logic to ensure only validly signed calls are processed.

Lumiterra: Resolved with `@26f722a66f ...`

Zenith: Verified.

[H-2] Survivor rewards distributed in current epoch become unclaimable for early vesting users

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LumiTownExtension.sol

Description:

When users stake their Agent tokens in `LaunchpadExtension`, they become eligible for **Survivor Rewards**, which are funded from the `buyHashCore` purchases made by other users. These rewards are distributed through the `advanceTownSurvivorReward` function, which allocates the available survivor pool balance among active stakers for the current epoch:

```
uint256 epoch = _getSurvivorRewardEpoch(block.timestamp);
...
uint256 _depositReward = Math.mulDiv(availableSurvivorPoolBalance,
    pointShare, PERCENT_DENOMINATOR);
...
bytes32 _agentTokenEpochHash = keccak256(abi.encodePacked(agentToken,
    epoch));
agentSurvivorRewardEpochInfo[_agentTokenEpochHash].totalReward
    += _depositReward;
```

Users later claim their share of these rewards through `createVestingSurvivorReward`, which creates vesting entries for all past undistributed epochs using:

```
uint256 _startEpoch = _lastCreatedEpoch + 1;
for (uint256 e = _startEpoch; e <= epoch; e++) { ... }
```

However, if a user already creates their vesting entries for epoch `X`, and then `advanceTownSurvivorReward` is called again during the same epoch `X`, the new rewards are added to the same epoch but cannot be claimed by that user later, since subsequent calls to `createVestingSurvivorReward` start from `X + 1`.

This effectively locks a portion of rewards in the current epoch, leaving them unclaimable.

Recommendations:

Consider either:

- Distributing rewards into the next epoch:

```
uint256 epoch = _getSurvivorRewardEpoch(block.timestamp) + 1;
```

- Disallowing vesting creation to include the current epoch:

```
uint256 _startEpoch = _lastCreatedEpoch + 1;
for (uint256 e = _startEpoch; e < epoch; e++) { ... }
```

Lumiterra: Resolved with `@e99ec37bf1 ...`

Zenith: Verified.

[H-3] lastUserVestingCreatedEpoch may be overwritten skipping vestings creation for previous epochs

SEVERITY: High	IMPACT: High
STATUS: Resolved	LIKELIHOOD: High

Target

- LaunchpadExtension.sol
- LumiTownExtension.sol

Description:

When a user stakes through `LumiTownExtension`, the function `LumiTownExtension.setStakeAgentBaseEpoch()` is called to initialize `lastUserVestingCreatedEpoch[user][agentToken]`:

```
function setStakeAgentBaseEpoch(
    address agentToken,
    address user
) external onlyLaunchpadExtension {
    uint256 epoch = _getSurvivorRewardEpoch(block.timestamp);
    lastUserVestingCreatedEpoch[user][agentToken] = epoch - 1;
}
```

This value is later used in `createVestingSurvivorReward()` function, which loops from `lastUserVestingCreatedEpoch[user][agentToken]` up to the current epoch to generate vesting rewards for each epoch.

However, if a user stakes **again before calling** `createVestingSurvivorReward()` is called, the `lastUserVestingCreatedEpoch[user][agentToken]` value will be **overwritten** with a newer epoch, causing vesting rewards from skipped epochs to be **lost permanently**.

Recommendations:

To prevent overwriting and loss of vesting epochs:

- Ensure `createVestingSurvivorReward()` is called for the user **before** updating `lastUserVestingCreatedEpoch[user][agentToken]`.

- Because the current function operates only on `msg.sender`, a minor redesign may be required — for example, allowing it to process vesting for a specified user safely.

Lumiterra: Resolved with `@8c6f57db71 ...`

Zenith: Verified.

4.2 Medium Risk

A total of 4 medium risk findings were identified.

[M-1] Incorrect reward rollback logic, leading to inaccurate survivor reward distribution

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LumiTownExtension.sol

Description:

Stakers earn "survivor pool" rewards that are **registered per-epoch** via `advanceTownSurvivorReward` (credited into `agentSurvivorRewardEpochInfo[epochHash].totalReward`). When a staker later creates vestings, the implementation **pre-reserves** their slice by incrementing `claimedReward` for that epoch.

```
uint256 _agentTotalReward
    = agentSurvivorRewardEpochInfo[_agentTokenEpochHash].totalReward;
uint256 _share = Math.mulDiv(_agentTotalReward,
    _userStakePosition.stakeAmount, _totalAgentStakeAmount);
```

On unstake, `_rollbackVestingByAgentToken` is meant to **free the unvested remainder** so it can be redistributed in the next `advanceTownSurvivorReward` call. However, the function currently subtracts the remaining rewards from both `claimedReward` **and** `totalReward`:

```
agentSurvivorRewardEpochInfo[h].claimedReward -= remainingReward;
agentSurvivorRewardEpochInfo[h].totalReward -= remainingReward;
```

This is wrong, because all the rewards were gone, part gone to the user and the other to the next `advanceTownSurvivorReward` call.

This would inflate `_agentTotalReward` when new users create vesting entries, and early vesters would unfairly get more rewards.

Proof of Concept:

```
function test_WrongDecreasedAmount() public {
    address user_1 = address(0x1234);
    address user_2 = address(0x12344);
    address user_3 = address(0x123444);

    pairedToken.mint(user_1, 100e18);
    pairedToken.mint(user_2, 100e18);
    pairedToken.mint(user_3, 100e18);

    ILaunchpad.DeploymentConfig memory cfg = _buildConfig(0, 0);

    pairedToken.approve(address(launchpad),
    cfg.agentLaunchConfig.createAgentFee);
    address agentToken = launchpad.deployToken(
        ILaunchpad.DeploymentBaseConfig({
            agentName: cfg.agentName,
            agentSymbol: cfg.agentSymbol,
            agentHash: cfg.agentHash
        })
    );

    PoolKey memory poolKey
    = launchpad.tokenDeploymentInfo(agentToken).poolKey;
    bytes memory hookData = abi.encode(
        IHook.PoolSwapData({
            mevModuleSwapData: bytes(""),
            poolExtensionSwapData: abi.encode(IHook.SwapHookData({user:
    address(this), extra: hex""}))
        })
    );

    vm.startPrank(user_1);
    pairedToken.approve(address(swapRouter), type(uint256).max);
    BalanceDelta swapDelta = swapRouter.swapTokensForExactTokens({
        amountOut: 2e18,
        amountInMax: type(uint256).max,
        zeroForOne: Currency.unwrap(poolKey.currency0) ==
    address(pairedToken),
        poolKey: poolKey,
        hookData: hookData,
        receiver: user_1,
        deadline: block.timestamp + 2
    });
    vm.stopPrank();
```

```
vm.startPrank(user_2);
pairedToken.approve(address(swapRouter), type(uint256).max);
swapDelta = swapRouter.swapTokensForExactTokens({
    amountOut: 2e18,
    amountInMax: type(uint256).max,
    zeroForOne: Currency.unwrap(poolKey.currency0) ==
address(pairedToken),
    poolKey: poolKey,
    hookData: hookData,
    receiver: user_2,
    deadline: block.timestamp + 2
});
vm.stopPrank();

vm.startPrank(user_3);
pairedToken.approve(address(swapRouter), type(uint256).max);
swapDelta = swapRouter.swapTokensForExactTokens({
    amountOut: 2e18,
    amountInMax: type(uint256).max,
    zeroForOne: Currency.unwrap(poolKey.currency0) ==
address(pairedToken),
    poolKey: poolKey,
    hookData: hookData,
    receiver: user_3,
    deadline: block.timestamp + 2
});
vm.stopPrank();

hook.claimProtocolFees(poolKey);

pairedToken.approve(address(lumiTownExtension), type(uint256).max);
lumiTownExtension.buyHashCore(
    ILumiTownExtension.BuyHashCoreSignature({
        buyer: address(this),
        salt: 0,
        deadline: block.timestamp,
        willCapKeys: new ILumiTownExtension.SettleCappedKeys[](0)
    }),
    10,
    ""
);

_advanceTownSurvivorReward(agentToken);

vm.startPrank(user_1);
Agent(agentToken).approve(address(launchpadExtension),
```

```
    Agent(agentToken).balanceOf(user_1));
    launchpadExtension.stakeAgentToken(agentToken,
    Agent(agentToken).balanceOf(user_1));
    vm.stopPrank();

    vm.startPrank(user_2);
    Agent(agentToken).approve(address(launchpadExtension),
    Agent(agentToken).balanceOf(address(user_2)));
    launchpadExtension.stakeAgentToken(agentToken,
    Agent(agentToken).balanceOf(address(user_2)));
    vm.stopPrank();

    vm.startPrank(user_3);
    Agent(agentToken).approve(address(launchpadExtension),
    Agent(agentToken).balanceOf(address(user_3)));
    launchpadExtension.stakeAgentToken(agentToken,
    Agent(agentToken).balanceOf(address(user_3)));
    vm.stopPrank();

    vm.prank(user_1);
    lumiTownExtension.createVestingSurvivorReward();

    vm.warp(block.timestamp + 4);

    assertEq(lumiTownExtension.getVestedRewardAmount(user_1),
    333333333333333333); // 0.333e18

    vm.prank(user_1);
    launchpadExtension.unStakeAgentToken(agentToken);

    vm.prank(user_2);
    lumiTownExtension.createVestingSurvivorReward();

    assertEq(lumiTownExtension.getVestedRewardAmount(user_2),
    334444444444444444); // 0.334e18

    vm.prank(user_3);
    lumiTownExtension.createVestingSurvivorReward();

    assertEq(lumiTownExtension.getVestedRewardAmount(user_3),
    332222222222222223); // 0.332e18
}
```

Recommendations:

Consider passing the `removedVestingAmount` to the `_rollbackVestingByAgentToken` function and decreasing the total rewards amount, not just the remaining:

```
agentSurvivorRewardEpochInfo[h].claimedReward -= remainingReward;
agentSurvivorRewardEpochInfo[h].totalReward -= remainingReward;
agentSurvivorRewardEpochInfo[h].claimedReward -= removedVestingAmount;
agentSurvivorRewardEpochInfo[h].totalReward -= removedVestingAmount;
```

Lumiterra: Resolved with `@fb4a7adc0c ...`

Zenith: Verified.

[M-2] Dividend share can be diluted by capped (inactive) keys during fee injections

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LumiTownExtension.sol

Description:

When users buy HashCore keys they join the active set for the epoch and earn per-share dividends from later buyers and from subsequent injections (e.g., trading fees). Each key also has a `maxRevenue`; once reached, the key should be considered capped/inactive and removed from `activeKeys[epoch]`. That pruning normally happens when the caller supplies `willCapKeys` and `_settleCappedKeys` is invoked during other flows (e.g., buys/claims).

However, `injectDividendRewardFromTradingFee` computes per-share using the current length of `activeKeys[coreRushEpoch]` without ensuring capped keys have been pruned:

```
uint256 activeKeysLength = activeKeys[coreRushEpoch].length;
uint256 deltaPerShare = Math.mulDiv(dividendAmount, ACC_SCALE,
    activeKeysLength);
globalAccPerShare[coreRushEpoch] += deltaPerShare;
```

If `activeKeys[epoch]` still includes capped keys, the divisor is too large, diluting dividends for truly active holders. A later buy path may partially compensate by redistributing "excess" when capped keys are finally settled:

```
uint256 remaining = activeKeys[coreRushEpoch].length;
uint256 extraDeltaPerShare = Math.mulDiv(totalExcessAmount, ACC_SCALE,
    remaining);
globalAccPerShare[coreRushEpoch] += extraDeltaPerShare;
```

But if no new buys occur, or if a user claims before the redistribution, they are paid from an under-credited `globalAccPerShare`, resulting in lower rewards.

Recommendations:

Consider either calling `_settleCappedKeys` at the very start of `injectDividendRewardFromTradingFee`, or adding a new public function that could be called whenever to prune these users. In addition to that, consider validating that the Hashcore's epoch is the same as the provided epoch in `_settleCappedKeys`.

Lumiterra: Resolved with `@14137af367 ...` In extreme cases where `willCapKeys` misses some expired keys, it is considered an acceptable risk.

Zenith: Verified.

[M-3] No ability to change feeManager

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- Hook.sol

Description:

The `Hook.setFeeManager()` function is restricted by the `onlyFactory` modifier, meaning that it can only be called by the `Launchpad` (factory) contract.

However, the `Launchpad` contract does **not** expose any functionality that allows calling `Hook.setFeeManager()`. As a result, once the fee manager is initialized during deployment, it **cannot be updated** later — limiting the protocol's ability to upgrade or change fee management logic if needed.

Recommendations:

Add a function to the `Launchpad` contract that allows authorized roles (e.g., owner or admin) to update the fee manager by calling `Hook.setFeeManager()` on the target hook contract.

Lumiterra: Resolved with `@75f2a10068 ...`

Zenith: Verified.

[M-4] Lack of slippage protection forces users to overpay for Hashcore purchases

SEVERITY: Medium	IMPACT: Medium
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LumiTownExtension.sol

Description:

The price of Hashcores increases each time new purchases occur. When a user initiates a purchase, they specify only the number of keys to buy, and the contract internally computes the total cost and transfers the required funds:

```
(uint256[] memory prices, uint256 totalCost)
    = _quotePerKeyPrices(_getGlobalKeyAmount(), purchaseKeyAmount);
// ...
CommonExtension.safeTransferFrom(baseToken, user, address(this), totalCost);
```

However, this mechanism lacks **slippage protection**.

If a user grants infinite approval to the extension and attempts to buy `X` keys expecting to pay a total of `Y`, another user could purchase additional keys (`Z`) just before the transaction is mined. As a result, the total cost (`totalCost`) increases, and the user ends up paying **more than expected** without any safeguard.

This can lead to unexpected loss of funds for buyers due to price changes between transaction submission and execution.

Recommendations:

Consider introducing a `maxTotalCost` parameter in the `buyHashCore` function to enforce an upper bound on the amount a user is willing to spend. The transaction should revert if the calculated cost exceeds this limit:

```
require(totalCost <= maxTotalCost, "Cost exceeds user limit");
```

This ensures users are protected from front-running or sudden price jumps between signing and execution.

Lumiterra: Resolved with `@f110124038 ...`

Zenith: Verified.

4.3 Low Risk

A total of 6 low risk findings were identified.

[L-1] Outdated epoch data returned by getters when `coreRushEpoch` end time has passed

`SEVERITY:` Low	`IMPACT:` Low
`STATUS:` Resolved	`LIKELIHOOD:` Medium

Target

- LumiTownExtension.sol

Description:

The protocol operates across distinct `coreRushEpochs`. Each epoch represents a period where players can buy Hashcores and compete for a jackpot. When a new epoch begins, key sales, cumulative prices, and jackpot rewards reset. However, `coreRushEpoch` only advances when `_initCoreRushEpoch()` is explicitly called.

Several view functions rely on `coreRushEpoch` to return reward or price information, such as:

- `getIndexRewardPool` — returns the jackpot reward for the current epoch
- `getGlobalAccPerShare` — returns the accumulated dividend per share for the epoch
- `quoteBatchHashCoreCost` — computes the cost of purchasing multiple keys

These functions reference mappings indexed by `coreRushEpoch` without validating whether the current epoch has already expired. For example:

```
function _getGlobalKeyAmount() internal view returns (uint256) {
    return globalKeyAmount[coreRushEpoch];
}
```

If the `currentCoreRushEndTime` has passed but `_initCoreRushEpoch()` has not yet been invoked, this will return outdated data for a completed epoch, potentially leading to incorrect pricing or reward calculations.

Recommendations:

Consider adding a time check to ensure that epoch-dependent getters handle expired epochs correctly. For example:

```
function _getGlobalKeyAmount() internal view returns (uint256) {
    uint256 currentCoreRushEndTime
    = finalJackpotReward[coreRushEpoch].endTime;
    if (currentCoreRushEndTime ≠ 0 && currentCoreRushEndTime <
    block.timestamp) {
        return 0;
    }
    return globalKeyAmount[coreRushEpoch];
}
```

Lumiterra: Resolved with @078afef99d ...

Zenith: Verified.

[L-2] LumiTownExtension allows keeper to provide totalPointShare not equal to 100%

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Low

Target

- LumiTownExtension.sol

Description:

When the keeper calls `LumiTownExtension.advanceTownSurvivorReward()`, they provide a `pointShare` value for each agent. The function then sums all of these and performs a validation against `PERCENT_DENOMINATOR`:

```
if (totalPointShare < (PERCENT_DENOMINATOR - 100000000) || totalPointShare >
    PERCENT_DENOMINATOR) revert Invalid();
```

This check allows `totalPointShare` to be **up to 1% lower** than `PERCENT_DENOMINATOR`. As a result, a portion of the `availableSurvivorPoolBalance` may remain undistributed, leading to unintended leftover funds.

Recommendations:

To ensure the full reward pool is distributed as intended, require that `totalPointShare` is **strictly equal** to `PERCENT_DENOMINATOR`:

```
if (totalPointShare < (PERCENT_DENOMINATOR - 100000000)
  || totalPointShare > PERCENT_DENOMINATOR) revert Invalid();
if (totalPointShare ≠ PERCENT_DENOMINATOR) revert Invalid();
```

This guarantees that the entire `availableSurvivorPoolBalance` is allocated precisely among agents.

Lumiterra: Resolved with @3601662303 ...

Zenith: Verified.

[L-3] Missing EIP-712 `version` in domain separator allows cross-upgrade Signature

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- BuyHashCoreProof.sol

Description:

The EIP-712 signature is intended to restrict buying Hashcores to only buyers with valid signatures approved by `townConfig.signerAddress`. The domain separator currently binds signatures to `(chainId, verifyingContract)`.

The domain separator omits a `version` field:

```
function _getDomainSeparator(address verifyingContract)
    private view returns (bytes32) {
    return keccak256(abi.encode(EIP712_DOMAIN_TYPEHASH, getChainId(),
    verifyingContract));
}
```

Because an upgradeable proxy keeps the same address, the pair `(chainId, verifyingContract)` does not change across implementations. As a result, signatures produced against the old implementation remain valid after an upgrade (same domain, same typehashes, same signer).

Recommendations:

Consider introducing a version string to the domain separator, and make sure to change/inrement it with every upgrade.

Lumiterra: Resolved with `@bf3a30dc7d ...`

Zenith: Verified.

[L-4] Epoch cannot advance or jackpot be claimed at exact `endTime` boundary

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LumiTownExtension.sol

Description:

At the exact boundary second when `block.timestamp == endTime`, the current logic creates a dead-zone where the epoch does not advance and the jackpot cannot be claimed:

- Epoch rotation only occurs when `endTime < block.timestamp`.
- Claiming reverts when `endTime ≥ block.timestamp`.

This means at equality (=) neither path succeeds: `_initCoreRushEpoch` won't advance, and `claimJackpotReward` reverts.

Recommendations:

Consider making the two conditions complementary so there is **no gap at equality**.

Lumiterra: Resolved with `@c3bff8c3f3 ...`

Zenith: Verified.

[L-5] Vesting entries are never pruned after claim

SEVERITY: Low	IMPACT: Low
STATUS: Resolved	LIKELIHOOD: Medium

Target

- LumiTownExtension.sol

Description:

Fully-vested entries are not removed when users claim. In `claimVestingSurvivorReward` → `_claimVestingInternal`, each vesting's claimed field is updated, but entries with `claimed == amount` remain in `userVestings[user]`. Vestings are only cleared during `removeVestingByAgentToken` (upon unstake).

Recommendations:

Consider pruning fully-claimed entries after updating them, in `_claimVestingInternal`.

Lumiterra: Resolved with `@104368c273 ...`

Zenith: Verified.

[L-6] Missing `IERC165` support in `supportsInterface()`

`SEVERITY:` Low	`IMPACT:` Medium
`STATUS:` Resolved	`LIKELIHOOD:` Low

Target

- LPLocker.sol
- MevModule.sol
- Hook.sol
- LumiTownExtension.sol
- LaunchpadExtension.sol
- FeeManagerExtension.sol

Description:

Several contracts implement a custom `supportsInterface` function that overrides the ERC-165 interface but fail to return support for the standard ERC-165 interface identifier (`0x01ffc9a7`).

According to the ERC-165 specification, a contract must return `true` for `0x01ffc9a7` in order to be recognized as ERC-165-compliant.

Omitting this value means external tools and other contracts cannot reliably detect the contract's supported interfaces.

Recommendations:

Consider updating the `supportsInterface` implementation to include the ERC-165 identifier.

Lumiterra: Resolved with `@b68fbc8ba7 ...`

Zenith: Verified.

4.4 Informational

A total of 6 informational findings were identified.

[I-1] globalStakeData.globalTotalStakeAmount mixes different agent tokens

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LaunchpadExtension.sol

Description:

When a user stakes agent tokens, the staked amount is added to the shared variable `globalStakeData.globalTotalStakeAmount`. However, this variable aggregates all agent tokens into a single value, making it non-informative and potentially misleading.

Recommendations:

It is recommended to remove the variable.

Lumiterra: Resolved with `@ef54cddccf ...`

Zenith: Verified.

[I-2] `placeLiquidity` returns a single `positionId` even when minting multiple LP positions

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LPLocker.sol

Description:

`ClankerLpLockerMultiple.placeLiquidity` delegates to `_mintLiquidity`, which can mint multiple LP positions in a single call (loop over `lockerConfig.tickLower.length`). However, both functions return only a single `uint256 positionId`:

```
// placeLiquidity
return _mintLiquidity(poolConfig, lockerConfig, poolKey, poolSupply, token);

// _mintLiquidity
positionId = positionManager.nextTokenId();
positionManager.modifyLiquidities(...);
```

Because several `Actions.MINT_POSITION` operations can be encoded and executed, callers cannot inspect the return value to determine all minted position IDs.

Recommendations:

Consider changing the return type of both `placeLiquidity` and `_mintLiquidity` to `uint256[] memory positionIds` and collect each minted NFT ID in the loop.

Lumiterra: Resolved with `@db39d1b0fa ...`

Zenith: Verified.

[I-3] Changing `refreshPeriod` without resetting schedule skews next refresh time

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- FeeManagerExtension.sol

Description:

`setCoreRushPoolRefreshPeriod` updates the settlement cadence but **does not** reset or realign `coreRushPoolReward.lastSettleTime`:

```
function setCoreRushPoolRefreshPeriod(uint256 refreshPeriod_)
    external onlyOwner {
    require(refreshPeriod_ > 0, "invalid refresh period");
    coreRushPoolReward.refreshPeriod = refreshPeriod_;
    emit RefreshPeriodUpdated(refreshPeriod_);
}
```

Because `_canRefresh(lastTime, refreshPeriod)` compares `block.timestamp` to the **previous** `lastSettleTime` using the **new** `refreshPeriod`, the next distribution can drift either waiting longer or shorter than expected, depending on whether the refresh period is increased or decreased.

Recommendations:

Consider setting `lastSettleTime` to `block.timestamp` in `setCoreRushPoolRefreshPeriod`, so a new cadence is started after resetting the refresh period.

Lumiterra: Resolved with `@1147a61a9f ...`

Zenith: Verified.

[I-4] Unnecessary `payable` on `deployToken`

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Launchpad.sol

Description:

`deployToken` is marked `payable`, but the create fee is **pulled in as ERC-20** via `SafeERC20.safeTransferFrom`:

```
IERC20 baseToken = launchpadConfig.getBaseToken();
_addCreateAgentFee(_createFee);
SafeERC20.safeTransferFrom(baseToken, msg.sender, address(this),
    _createFee);
```

The function never reads `msg.value`, making the `payable` flag unnecessary.

Recommendations:

Consider removing `payable`.

Lumiterra: Resolved with `@f4b22f6d84 ...`

Zenith: Verified.

[I-5] Unnecessary `< 0` Check in `buyHashCore` Function

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- LumiTownExtension.sol

Description:

The `buyHashCore` function allows users to purchase hash core keys by passing a `purchaseKeyAmount` (of type `uint256`). However, the validation check currently includes an impossible condition:

```
if (purchaseKeyAmount < 0 || purchaseKeyAmount > 2000) revert Invalid();
```

Since `purchaseKeyAmount` is an unsigned integer (`uint256`), the `< 0` branch is never reachable and has no effect.

Recommendations:

Consider refactoring the condition to accurately capture invalid values, ensuring zero or excessive purchases are properly restricted:

```
if (purchaseKeyAmount < 0 || purchaseKeyAmount > 2000) revert Invalid();
if (purchaseKeyAmount == 0 || purchaseKeyAmount > 2000) revert Invalid();
```

Lumiterra: Resolved with `@8f980ad3e3 ...`

Zenith: Verified.

[I-6] Remove unused parameter

SEVERITY: Informational	IMPACT: Informational
STATUS: Resolved	LIKELIHOOD: Low

Target

- Hook.sol

Description:

`Hook._deltaOnSpecifiedLeg()` function has `zeroForOne` param that is not used by the logic.

Recommendations:

Remove unused parameter.

Lumiterra: Resolved with `@4d9547cfa0 ...`

Zenith: Verified.