Three Sigma Labs

Code Audit

leNFT

leNFT Trading & Lending Protocol

Disclaimer

Code Audit

leNFT Trading & Lending Protocol

Disclaimer

The ensuing audit offers no assertions or assurances about the code's security. It cannot be deemed an adequate judgment of the contract's correctness on its own. The authors of this audit present it solely as an informational exercise, reporting the thorough research involved in the secure development of the intended contracts, and make no material claims or guarantees regarding the contract's post-deployment operation. The authors of this report disclaim all liability for all kinds of potential consequences of the contract's deployment or use. Due to the possibility of human error occurring during the code's manual review process, we advise the client team to commission several independent audits in addition to a public bug bounty program.

Table of Contents

Code Audit

IeNFT Trading & Lending Protocol

Table of Contents

Summary

Code Audit

IeNFT Trading & Lending Protocol

Summary

Three Sigma Labs audited leNFT in a 6 person week engagement. The audit was conducted from 05-06-2023 to 23-06-2023.

Protocol Description

leNFT is a protocol that aims to revolutionize NFT finance by providing a robust and efficient platform for NFT trading and lending.

For trading, the protocol utilizes an Automated Market Maker (AMM) model based on the lssvm model popularized by sudoswap. This approach aims to provide the most efficient liquidity utilization, resulting in more profits for liquidity providers and better pricing for traders. This makes leNFT an ideal choice for traders looking for low slippage and high capital efficiency.

For lending, leNFT employs a peer-to-pool lending architecture that allows NFT holders to access instant liquidity by borrowing against their assets. Liquidity providers can deposit into lending pools and collect rewards originating from the loans' interest payments. This incentivizes liquidity providers to participate in the ecosystem, and the borrowers can access instant liquidity without having to sell their NFTs.

leNFT also features a vote-gauge system, similar to that of Curve, which incentivizes liquidity providers by distributing LE inflation to LP providers through the use of Gauges. NFT projects are incentivized to lock veLE in order to provide liquidity within their ecosystems. [1]

Scope

Code Audit

IeNFT Trading & Lending Protocol

Scope

All files present in the contracts folder.

```
Java
contracts
├── libraries
│   ├── balancer
│   │   └── ERC20Helpers.sol
│   ├── logic
│   │   ├── BorrowLogic.sol
│   │   └── LiquidationLogic.sol
│   ├── types
│   │   ├── ConfigTypes.sol
│   │   └── DataTypes.sol
│   └── utils
│       ├── PercentageMath.sol
│       └── SafeCast.sol
└── protocol
    ├── AddressProvider.sol
    ├── Bribes.sol
    ├── FeeDistributor.sol
    ├── Gauges
    │   ├── GaugeController.sol
    │   ├── LendingGauge.sol
    │   └── TradingGauge.sol
    ├── GenesisNFT.sol
    ├── Lending
    │   ├── InterestRate.sol
    │   ├── LendingMarket.sol
    │   ├── LendingPool.sol
    │   ├── LoanCenter.sol
    │   ├── NFTOracle.sol
    │   └── TokenOracle.sol
    ├── NativeToken.sol
    ├── NativeTokenVesting.sol
    ├── Trading
    │   ├── LiquidityPairMetadata.sol
    │   ├── PricingCurves
    │   │   ├── Exponential.sol
    │   │   └── Linear.sol
    │   ├── SwapRouter.sol
    │   ├── TradingPool.sol
    │   ├── TradingPoolFactory.sol
```

```
|   └── TradingPoolHelpers.sol
├── Trustus
|   └── Trustus.sol
├── VotingEscrow.sol
└── WETHGateway.sol
```

The review was conducted on the code present in the leNFT public repository, which contains the main contracts, testing scripts as well as a documentation providing additional information. The code was frozen for review at commit 78170402e176f2f754e0e24cb22a90961c9e5799 .

Assumptions

The scope of the audit was carefully defined to include the contracts at the lowest level of the inheritance hierarchy, as these are the ones that will be deployed to the mainnet. The only external libraries used in the implementation of these contracts were ones trusted by the community (i.e. OpenZeppelin, Balancer and AAVE) - these libraries have already been battle-tested by multiple protocols, guaranteeing a high level of security.

Methodology

Code Audit

leNFT Trading & Lending Protocol

Methodology

To begin, we reasoned meticulously about the contract's business logic, checking security-critical features to ensure that there were no gaps in the business logic and/or inconsistencies between the aforementioned logic and the implementation. Second, we thoroughly examined the code for known security flaws and attack vectors. Finally, we discussed the most catastrophic situations with the team and reasoned backwards to ensure they are not reachable in any unintentional form.

Taxonomy

In this audit we report our findings using as a guideline Immunefi's vulnerability taxonomy, which can be found at immunefi.com/severity-updated/. The final classification takes into account the severity, according to the previous link, and difficulty of the exploit. The following table summarizes the general expected classification according to severity and difficulty; however, each issue will be evaluated on a case-by-case basis and may not strictly follow it.

Severity / Difficulty	HIGH	MEDIUM	LOW
NONE	None		
LOW	Low		
MEDIUM	Low	Medium	Medium
HIGH	Medium	High	High
CRITICAL	High	Critical	Critical

Project Dashboard

Code Audit

IeNFT Trading & Lending Protocol

Project Dashboard

Application Summary

Name	leNFT
Commit	7817040
Language	Solidity
Platform	Ethereum

Engagement Summary

Timeline	05 June to 23 June, 2023
№ of Auditors	2
Review Time	6 person weeks

Vulnerability Summary

Issue Classification	Found	Addressed	Acknowledged
Critical	1	1	0
High	1	1	0
Medium	2	2	0
Low	5	5	0
None	26	26	0

Category Breakdown

Suggestion	9
Documentation	1
Bug	4
Optimization	20
Good Code Practices	3

Code Maturity Evaluation

Code Audit

leNFT Trading & Lending Protocol

Code Maturity Evaluation

Code Maturity Evaluation Guidelines

Category	Evaluation
Access Controls	The use of robust access controls to handle identification and authorization and to ensure safe interactions with the system.
Arithmetic	The proper use of mathematical operations and semantics.
Centralization	The presence of a decentralized governance structure for mitigating insider threats and managing risks posed by contract upgrades
Code Stability	The extent to which the code was altered during the audit.
Upgradeability	The presence of parameterizations of the system that allow modifications after deployment.
Function Composition	The functions are generally small and have clear purposes.
Front-Running	The system's resistance to front-running attacks.
Monitoring	All operations that change the state of the system emit events, making it simple to monitor the state of the system. These events need to be correctly emitted.
Specification	The presence of comprehensive and readable codebase documentation.
Testing and Verification	The presence of robust testing procedures (e.g., unit tests, integration tests, and verification methods) and sufficient test coverage.

Code Maturity Evaluation Results

Category	Evaluation
Access Controls	Satisfactory. The codebase has a strong access control mechanism.
Arithmetic	Satisfactory. The codebase uses Solidity version >0.8.0 as well as takes the correct measures in rounding the results of arithmetic operations.
Centralization	Weak. The owner has significant privileges over the protocol, namely upgrading the contracts and setting new addresses using AddressProvider
Code Stability	Satisfactory. The code was stable during the audit.
Upgradeability	Satisfactory. Major contracts can be upgraded by the owner.
Function Composition	Satisfactory. There is little duplicated logic and functions have a clear purpose.
Front-Running	Satisfactory. There are little front-running opportunities
Monitoring	Moderate. Some events are emitted, however some lack info and some state changing operations are not accompanied by events. [3S-LENFT-L03, N06]
Specification	Moderate. There is documentation, but sometimes lacking or not properly updated.
Testing and Verification	Satisfactory.There is an adequate testing suite with unit, integration, functional and fuzz testing.

Findings

Code Audit

leNFT Trading & Lending Protocol

Findings

3S-LENFT-C01

protocol/FeeDistributor.sol: claiming rewards will not allow for future checkpoints

Id	3S-LENFT-C01
Classification	Critical
Severity	Critical
Likelihood	High
Category	Bug
Relevant Links	protocol/FeeDistributor.sol master/POC-3S-LENFT-C01.js

Description

The current implementation of the FeeDistributor requires an always increasing balance for correct operation. This happens since the **`_accountedFees[token]`** mapping can never be decreased. It is only updated in the following function:

```
function checkpoint(address token) external override {
    (...)
    // Find the current balance of the token in question
    uint256 balance = IERC20Upgradeable(token).balanceOf(address(this));
    // Add unaccounted fees to current epoch
    _epochFees[token][currentEpoch] += balance - _accountedFees[token];
    // Update total fees accounted for
    _accountedFees[token] = balance;
}
```

This logic works whenever the balance of the contract is increasing, but an issue arises when someone claims rewards, withdrawing funds from this contract without updating the **`_accountedFees`** mapping. If this ever happens, all subsequent calls to checkpoint() will most likely revert, since the result of **`balance - _accountedFees[token]`** will be negative, leading to an underflow.

Since the **`checkpoint()`** function is called whenever fees are transferred to the protocol, this function reverting will cause **`claimLiquidation()`** (on the Lending part of the protocol) and **`buy()`**/**`sell()`** (on the Trading side of the protocol) to also revert, not allowing users to claim liquidations nor buy/sell NFTs to the protocol.

Recommendation

In the **`claim()`** function, just before sending the rewards to the user, the **`_accountedFees`** mapping should be updated to account for this change in contract balance. This can be achieved with the following line: **`_accountedFees[token] -= amountToClaim;`**

Tests for this scenario should also be included in the tests folder.

POC

POC-3S-LENFT-C01.js

Status

Addressed here: leNFT/contracts@3a3a6bf

3S-LENFT-H01

protocol/WETHGateway.sol: contract can be griefed not allowing borrowing

Id	3S-LENFT-H01
Classification	High
Severity	High
Likelihood	Medium
Category	Bug
Relevant Links	protocol/WETHGateway.sol#L117

Description

The borrow operation of the WETHGateway contract can be rendered useless if someone sends 1 wei to the contract (causing it to always revert on line 117):
`assert(_weth.balanceOf(address(this)) == amount);`

This issue is not critical since the WETHGateway contract is just a router which simplifies operations with eth (by converting it to weth), however, the severity is still high since it is used by the front end, so this exploit could significantly harm user experience.

Recommendation

Remove this line since the **`market.borrow()`** call will either revert or transfer the amount to the contract. Even if it doesn't, the calls to unwrap the weth and send it back to the user will also revert, so the user funds are always protected.

Note: Is is always possible to remove the balance of the WETHGateway contract, since anyone can use **`depositTradingPool()`** to get approval for all NFTs and weth from the contract. This would not present a definitive solution though, as any user could send another wei to the contract, blocking it again.

Status

Addressed here: leNFT/contracts@c2c3eaa

3S-LENFT-M01

protocol/Gauges/GaugeController.sol: incorrect getRewardsCeiling() logic

Id	3S-LENFT-M01
Classification	Medium
Severity	Medium
Likelihood	Medium
Category	Bug
Relevant Links	Gauges/GaugeController.sol#L478-L491

Description

The current implementation of the GaugeController has the following function to calculate the rewards ceiling per epoch:

```
function getRewardsCeiling(uint256 epoch) public view returns (uint256) {
    uint256 inflationEpoch;
    // If we are in the loading period, return smaller rewards
    if (epoch < LOADING_PERIOD) {
        return (_initialRewards * epoch) / LOADING_PERIOD;
    } else if (inflationEpoch > MAX_INFLATION_PERIODS) {
        inflationEpoch = MAX_INFLATION_PERIODS;
    } else {
        inflationEpoch = epoch / INFLATION_PERIOD;
    }
    return
        (_initialRewards * (3 ** inflationEpoch)) / (4 ** inflationEpoch);
}
```

The issue here is that the **`inflationEpoch`** is a memory variable initialized to zero, so the condition **`if (inflationEpoch > MAX_INFLATION_PERIODS)`** will always be false, and the code will always set the **`inflationEpoch = epoch / INFLATION_PERIOD`** after the initial loading period of 6 months. This results in an incorrect calculation of the RewardsCeiling after the **`MAX_INFLATION_PERIODS`** (8 years), where the RewardsCeiling will never reach its cap.

Recommendation

Fix the getRewardsCeiling function.

Note: Since the formula for the RewardsCeiling is not described in the documentation, a more specific suggestion can't be provided.

Status

Addressed here: leNFT/contracts@f8f48a7

3S-LENFT-M02

protocol/GenesisNFT.sol: Only 1336 GenesisNFT tokens can be minted

Id	3S-LENFT-M02
Classification	Medium
Severity	Medium
Likelihood	High
Category	Bug
Relevant Links	documentation protocol/GenesisNFT.sol#L42 protocol/GenesisNFT.sol#L344

Description

The documentation mentions that the Genesis Mint is limited to just 1337 tokens, which is corroborated by the constant set in the GenesisNFT contract: **`MAX_CAP = 1337;`**

The issue is that the current implementation of the GenesisNFT contract only allows for 1336 NFTs to be minted. This contract uses the following logic for minting:

```
contract testCap {
    uint256 constant MAX_CAP = 10;
    uint256 counter = 1;
    function mint(uint256 amount) external {
        require(counter + amount <= MAX_CAP, "G:M:CAP_EXCEEDED");

        for (uint256 i = 0; i < amount; i++) {
            // Mint
            counter ++;
        }
    }
}
```

Here, even though the **`MAX_CAP`** is set at 10 tokens, if you ever try to call the mint function with an amount of 10, the call will revert since the counter is initialized at 1, and so the require condition **`11 <= 10`** will be false.

Recommendation

Change line 344 to `_tokenIdCounter.current() + amount <= getCap() + 1,` and add a test for this situation

Status

Addressed here: leNFT/contracts@59e5b41

3S-LENFT-L01

Throughout code: functions are not reentrant secure

Id	3S-LENFT-L01
Classification	Low
Severity	Low
Likelihood	Low
Category	Suggestion
Relevant Links	Solidity Documentation

Description

There are several instances of functions that don't use the secure Checks-Effects-Interactions pattern commonly used in solidity to prevent reentrancy attacks. This pattern states that solidity functions should:

- start by checking the contract state and function arguments, making sure they match the required conditions.

- make all changes to internal storage as a result of the function call.

- leave for last external calls that could give flow control back to the user.

This practice ensures that all possible reentrant calls will be made after state changes have taken effect, making the reentrant calls identical to an ordinary call made in a future transaction (minimizing exposure to reentrancy attacks).

In the current implementation, this is not a severe issue since the nonReentrant modifier used is effective against most reentrancy attacks, however, this practice is still recommended since it:

- doesn't cost any gas (unlike mutexes which constantly require storage reads and writes)

- prevents reentrancy even in view functions (which don't usually have the nonReentrant modifier). This is useful since reentrancy could be performed in a different protocol or third party contract, which could use getters to obtain outdated or uninitialized storage values.

For instance, function **`createLock()`** on the VotingEscrow contract, mints ERC721 tokens to users (letting then take execution control with function onERC721Received) before all the appropriate storage initialization is performed, allowing users to call functions like

`claimRebates()` with a token that hasn't been initialized yet (i.e. `_tokenIdCounter` hasn't been incremented, `_lockedBalance[tokenId]` and `_nextClaimableEpoch[tokenId]` are still null, and the native tokens haven't been transferred to the contract).

Recommendation

For security redundancy, and to prevent future problems on upgrades or third party integration, the use of this pattern is heavily suggested.

Some detected examples include:

- **VotingEscrow:** function `createLock()`
- **Trading Gauge:** function `withdraw()`
- **Genesis NFT:** function `mint()`

Here, the lines that allow user reentrancy (usually calls that transfer or mint ERC217 tokens to users) should be placed at the very end of the respective functions.

Note: More information on this pattern can be found in the Solidity Documentation

---

Status

Addressed here: leNFT/contracts@f15aa0b

3S-LENFT-L02

Throughout codebase: cache variables to save on gas

Id	3S-LENFT-L02
Classification	Low
Severity	Low
Likelihood	Medium
Category	Optimization, Good Code Practices
Relevant Links	Trading/TradingPool.sol#L384-L390 Trading/TradingPool.sol#L493-L499 Trading/TradingPool.sol#L384-L390 Trading/TradingPool.sol#L493-L499 protocol/VotingEscrow.sol#L196-L200 Lending/LendingMarket.sol#L209-L212 Trading/TradingPool.sol#L384-L390 Trading/TradingPool.sol#L493-L499

Description

Throughout the codebase, several times storage variables are loaded and used in the same scope which incurs in significant gas costs since a SLOAD operation is much more expensive than a MSTORE and then consequent MLOAD operations. Below is a list of several times where this occurs:

TradingPool

- **`_addressProvider.getFeeDistributor()`** is called twice in the buy and sell functions.

- **`_addressProvider`** is loaded several times in the buy and sell functions.

VotingEscrow

- writeTotalWeightHistory, a lot of storage reads are made in this function, save the most used ones in memory

- **`getEpoch()`**: cache variable **`_deployTimestamp`** to prevent one storage read

- cache **`_addressProvider.getNativeToken()`** (lines 220 and 225)

Bribes:

- **`claim()`**: cache **`IVotingEscrow(votingEscrow).getEpoch(lockLastPoint.timestamp)`** to prevent an external call

- **`withdrawBribe()`**: cache **`_userBribes[token][gauge][nextEpoch][msg.sender]`**

LoanCenter

- **`liquidateLoan()`** cache **`_loans[loanId].owner`**

Router

- cache **`_addressProvider.getTradingPoolFactory()`**

Fee Distribuitor

- cache **`_epochFees[token][epoch]`** in **`salvageFees()`**

- cache **`votingEscrow.getEpoch(block.timestamp)`** (line 146)

- cache **`IERC721Upgradeable(address(votingEscrow)).ownerOf(tokenId)`** in **`claim()`**

Genesis NFT:

- cache **`_addressProvider.getWETH()`** (lines 367 and 374 and 394)

- cache **`_addressProvider.getVotingEscrow()`** (lines 433 and 437)

- cache **`_tokenIdCounter.current()`**

- cache **`_addressProvider.getNativeToken()`** and **`_addressProvider.getWETH()`** (line 654 and 655)

GaugeController

- cache **`getTotalWeightAt(epoch)`** function **`getGaugeRewards()`**

Lending Gauge

- cache **`votingEscrow.getEpoch(block.timestamp)`** (line 101)

TradingGauge

- cache **`_lpValue[lpId]`** (lines 348 and 349)

LendingMarket

- createLendingPool() → **`_addressProvider`** is loaded from storage several times. **`_poolCount`** is also loaded a couple times

WETHGateway

- throughout the contract: _weth variable should be cached in memory in each function (and other storage variables if possible), since it is used more than 2 times most times.

Recommendation

We understand that sometimes saving variables from storage to memory is not possible due to the possibility of the "Stack too deep" error; nevertheless, this pattern should still be followed whenever possible. Below is an example implementation in TradingPool, where the fee distributor is loaded twice from storage in the same function both in the buy function and in the sell function.

```
address feeDistributor_ = _addressProvider.getFeeDistributor();
IERC20(_token).safeTransfer(feeDistributor_ ,
PercentageMath.percentMul(totalFee, protocolFeePercentage));
IFeeDistributor(feeDistributor_ ).checkpoint(_token);
```

Status

Addressed here: leNFT/contracts@9e0e7dc

3S-LENFT-L03

protocol/Bribes.sol: WithdrawBribe event should be emitted with the msg.sender

Id	3S-LENFT-L03
Classification	Low
Severity	Low
Likelihood	Low
Category	Suggestion
Relevant Links	

Description

At the moment, the Bribes contract implements the following logic to deposit and withdraw bribes:

```
function depositBribe(address briber, address token, address gauge, uint256
amount){
    _userBribes[token][gauge][nextEpoch][briber] += amount;
    IERC20Upgradeable(token).safeTransferFrom(msg.sender, address(this),
amount);
    emit DepositBribe(briber, token, gauge, amount);
}
function withdrawBribe(address receiver, address token, address gauge,
uint256 amount){
    _userBribes[token][gauge][nextEpoch][msg.sender] -= amount;
    IERC20Upgradeable(token).safeTransfer(receiver, amount);
    emit WithdrawBribe(receiver, token, gauge, amount);
}
```

Here, the issue is that the withdrawBribe function removes the bribe from the mapping **`_userBribes`** corresponding to the "msg.sender", but emits the event with the "receiver" address. This means the user withdrawing the bribe in storage is not the one emitted by the event, which also conflicts with the DepositBribe event (where the address emitted is the one being changed in storage).

Recommendation

For consistency between the **`_userBribes`** mapping and DepositBribe/WithdrawBribe events, the WithdrawBribe event should be emitted as: **`emit WithdrawBribe(msg.sender, gauge, amount);`**.

Status

Addressed here: leNFT/contracts@ef492de

3S-LENFT-L04

Trading/TradingPool.sol: track totalProtocolFee instead of totalFee

Id	3S-LENFT-L04
Classification	Low
Severity	Low
Likelihood	Low
Category	Suggestion
Relevant Links	

Description

On the TradingPool contract, the buy() and sell() functions use the following logic for the protocol fee:

```
function buy()
    for (uint i = 0; i < nftIds.length; i++) {
        fee = spotPrice * fee;
        protocolFee = fee * protocolFeePercentage;
        tokenAmount += (spotPrice + fee - protocolFee);
        totalFee += fee;
    }
    protocoTotalFee = totalFee * protocolFeePercentage
}
```

This logic is not optimized gas-wise, and will lead to mathematical rounding errors, which result in a sum of protocolFee's subtracted from the LPs slightly smaller than the total fee sent to the protocol.

Note: These rounding errors could even not allow LPs to remove their liquidity, in an extreme case where the contract's balance would go negative by that tiny margin.

Recommendation

Since `totalFee` is only used to compute the `protocoTotalFee`, store the `totalProtocolFee` instead of `totalFee` to improve readability, save gas and protect against rounding errors, changing the above logic to:

```
function buy()
    for (uint i = 0; i < nftIds.length; i++) {
        fee = spotPrice * fee;
        protocolFee = fee * protocolFeePercentage;
        tokenAmount += (spotPrice + fee - protocolFee);
        totalProtocolFee += protocolFee ;
    }
}
```

Status

Addressed here: leNFT/contracts@407c469 and leNFT/contracts@4fd7404

3S-LENFT-L05

protocol/Lending/TokenOracle.sol: possibility of oracle prices rounding to zero

Id	3S-LENFT-L05
Classification	Low
Severity	Low
Likelihood	Low
Category	Suggestion
Relevant Links	Lending/TokenOracle.sol#L56

Description

On line 56 of the TokenOracle: **`return uint256(price) * (PRICE_PRECISION / feedPrecision);`** there is a possibility that **`(PRICE_PRECISION / feedPrecision)`** rounds down to zero, leading to a returned price of 0 eth per token.

At the moment this should not happen, since **`PRICE_PRECISION = 1e18`** and **`feedPrecision = 10 ** priceFeed.decimals()`**, with the current chainlink AggregatorV3Interface's decimals being set to 8.

There is, however, the possibility that the returned decimals change in a future update, or that the team changes the PRICE_PRECISION in a future version or the protocol.

Recommendation

Changing line 56 to **`return (uint256(price) * PRICE_PRECISION) / feedPrecision;`** is safer, since if the multiplication ever overflowed, there would be an error (and the team could solve the problem by setting a new oracle with a different **`PRICE_PRECISION`**). The issue of overflow is also a lot less likely, since a uint256s can store values up to around 1.16e+77.

Status

Addressed here: leNFT/contracts@3496906

3S-LENFT-N01

protocol/Trading/SwapRouter.sol: nonReentrant modifier isn't necessary

Id	3S-LENFT-N01
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	Trading/SwapRouter.sol protocol/WETHGateway.sol

Description

Since the SwapRouter contract is just a router (it doesn't hold any funds or storage variables), and the only user that can reenter is the caller, its external function does not need the nonReentrant security measure. Moreover, since the functions it is calling already have the same modifier, the exposure to reentrancy attacks is basically null.

Recommendation

Remove the nonReentrant modifier to save gas.

Note: This issue is also valid for the WETHGateway contract

Status

Addressed here: leNFT/contracts@7ce25ea

3S-LENFT-N02

libraries/types/DataTypes.sol: structs should be packed

Id	3S-LENFT-N02
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	types/DataTypes.sol

Description

Structs, especially when used in mappings and stored multiple times, should be as compact as possible, saving a considerable amount of gas on storage reads and writes.

Recommendation

The following structs, defined in the DataTypes library, should be packed, taking up less storage space:

- **`NftToLp`** (indexes can be uint128s)

- **`LiquidityPair`** (**`spotPrice`** and **`tokenAmount`** can be uint128s and **`fee`** and **`delta`** uint48)

- **`LockedBalance`** (**`amount`** can be a uint216 and the **`timestamp`** a uint40)

- **`WorkingBalance`** (**`timestamp`** can be stored as uint40 and all variables can be packed into a single storage slot)

- **`Point`** (**`timestamp`** can be stored as uint40 and all variables can be packed into a single storage slot)

- **`MintDetails`** (**`lpAmount`** should be uint176 or smaller)

Status

Addressed here: leNFT/contracts@a683386

3S-LENFT-N03

libraries/logic /BorrowLogic.sol: Move GenesisNFT validation logic to GenesisNFT contract

Id	3S-LENFT-N03
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	logic/BorrowLogic.sol#L218-L245

Description

In the current implementation, the logic to validate if a genesisNFT can be locked by the msg.sender on behalf of a user during loan creation is being executed on the BorrowLogic.sol contract. This requires 5 calls to the genesisNFT to validate the parameters and lock the genesisNFT.

Recommendation

All this logic could be placed inside a function **`lockGenesisNFT()`** inside the genesisNFT contract, preventing all those external calls. This function could be the first thing executed during the borrow validation and could also return the **`maxLTVBoost`** necessary for future validations. This change should therefore also involve the replacement of function **`setLockedState()`** with **`lockGenesisNFT()`** and **`unlockGenesisNFT()`**

Note: We are aware that the GenesisNFT contract is already quite large, so this optimization might not be possible to implement if it would cause the GenesisNFT contract to exceed the maximum contract size.

Status

Addressed here: leNFT/contracts@43354af

3S-LENFT-N04

libraries/logic /LiquidationLogic.sol: entire loan struct is being needlessly loaded

Id	3S-LENFT-N04
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	logic/LiquidationLogic.sol#L71

Description

In function **`bidLiquidationAuction()`** of the LiquidationLogic, the entire loan struct is being loaded from storage just to access **`loanData.state`** and **`loanData.pool`**. Since the loanCenter already has getters for these two variables, it is cheaper to use these two getters instead of calling the function **`loanCenter.getLoan(params.loanId)`**.

Recommendation

Use getters **`getLoanLendingPool()`** and **`getLoanState()`** instead of **`getLoan()`** to save gas on storage loads.

Status

Addressed here: leNFT/contracts@babafec

3S-LENFT-N05

libraries/types/ConfigTypes.sol: pack InterestRateConfig struct variables to save gas

Id	3S-LENFT-N05
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	types/ConfigTypes.sol#L24

Description

Currently, the InterestRateConfig struct has the following variables:

```
struct InterestRateConfig {
    uint256 optimalUtilizationRate;
    uint256 baseBorrowRate;
    uint256 lowSlope;
    uint256 highSlope;
}
```

These variables are either rates or slopes, with values in the order of magnitude of a few thousand on the test parameters. Since this struct is used quite frequently and most of the time all the parameters are loaded in the same transaction, packing all these variables into a single storage slot would save a great deal of gas.

Storing these 4 variables as uint64s would still allow them to hold values up to ~1.84e+19, far above the interval required for their application.

Recommendation

Change the struct variable's type from uint256 to uint64.

Status

Addressed here: leNFT/contracts@3339a7c and leNFT/contracts@9ad882c

3S-LENFT-N06

Throughout codebase: some events could be emitted

Id	3S-LENFT-N06
Classification	None
Severity	None
Likelihood	
Category	Suggestion
Relevant Links	Lending/InterestRate.sol#L32-L39 Lending/InterestRate.sol#L43-L48 Lending/LendingMarket.sol#LL258C1-L259C1

Description

- In InterestRate.sol, when adding or removing a token no event is emitted publicizing this change.

- When creating a pool here, more information could be given in the event.

Recommendation

- Emit "TokenAdded" and "TokenRemoved" events when adding or removing a token.

- Also emit the underlying asset of the pool in the **`CreateLendingPool`** event

Status

Addressed here: leNFT/contracts@e7bb474

3S-LENFT-N07

protocol/Lending/TokenOracle.sol: getTokenETHPrice() should also return price precision

Id	3S-LENFT-N07
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	Lending/TokenOracle.sol logic/ValidationLogic.sol#L147-L148 logic/ValidationLogic.sol#L215-L218

Description

In the current implementation of the TokenOracle contract, 2 external calls are necessary to get the real token eth price: one to function **`getTokenETHPrice()`** and another to function **`getPricePrecision()`** (to get the correct decimal places). This leads to unnecessary external calls in functions: validateBorrow() and validateCreateLiquidationAuction() in the ValidationLogic contract, resulting in gas waste.

Recommendation

In the TokenOracle contract, function getTokenETHPrice() should return a tuple with the price and the price precision, since the two are always necessary to get the actual token price.

Status

Addressed here: leNFT/contracts@c0775d1

3S-LENFT-N08

Documentation: Incorrect formula for rewards

Id	3S-LENFT-N08
Classification	None
Severity	None
Likelihood	
Category	Documentation
Relevant Links	Gauges Gauges/GaugeController.sol#L497

Description

On the Gauges documentation page, the formula for the **`rewards(epoch)`** presented does not match the plotted curve nor the code implementation.

The formula presented is:

- **`rewards = ceiling * (locked_LE / (5 * total_LE)) ^3`**

This represents a monotonically increasing function starting at 0 (when the locked tokens are 0) and reaching 0.008*ceiling (when the locked_LE is equal to the total_LE). This function does not match the decreasing function plotted in the graph.

The code implementation uses the formula:

- **`rewards = ceiling * (1 - (locked_LE / (5 * total_LE))) ^3`**

which starts at 1* ceiling and decreases to 0.512*ceiling, matching the plotted curve.

Recommendation

Fix the documentation

Status

Addressed here: leNFT/docs@de4ef25

3S-LENFT-N09

Throughout codebase: remove imports of unused libraries

Id	3S-LENFT-N09
Classification	None
Severity	None
Likelihood	
Category	Good Code Practices
Relevant Links	Lending/LendingMarket.sol#L4-L25 Lending/LoanCenter.sol#L4-L13 protocol/GenesisNFT.sol#L4-L27 Trading/TradingPoolFactory.sol#L4-L15

Description

In several files in the codebase, several libraries are imported but never used.

LendingMarket: ILoanCenter, ERC721Upgradeable

LoanCenter: IERC721Upgradeable, Trustus

GenesisNFT: Initializable

TradingPoolFactory: ERC721Upgradeable

Recommendation

Remove the mentioned imports.

Status

Addressed here: leNFT/contracts@c3b3f71

3S-LENFT-N10

Throughout codebase: overuse of libraries

Id	3S-LENFT-N10
Classification	None
Severity	None
Likelihood	
Category	Optimization, Good Code Practices
Relevant Links	

Description

The protocol uses several libraries to aid in some operations. We consider that this flow is overused and some libraries can be removed or reduced.

It should also be noted that the way the architecture is implemented right now, it sometimes makes code confusing and makes the code development more prone to errors in the future. For example, having validation logic separated makes it harder in certain function calls to check if the arguments have previously been validated or not and increases the amount of calls needed in the validation libraries since the information needed has to be fetched from other contracts. Moreover, this flow could also hurt future code upgrades.

Recommendation

- remove LockLogic library and incorporate the few lines of code in the contract it is used

- remove LoanLogic library and incorporate the few lines of code in the contract it is used

- as a general rule of thumb, we would recommend not using libraries (more expensive and makes the code confusing since it leads to several jumps between files) in situations where the code logic itself is only being used once;

Note: since the protocol is so far into code development we understand it is not feasible to make this change, but would advise against this architecture in future coding endeavors.

Status

Addressed here: leNFT/contracts@4c03415, leNFT/contracts@cc9c92c and leNFT/contracts@a739274

3S-LENFT-N11

protocol/Trading/LiquidityPairMetadata.sol: internal function "trait" should start with underscore

Id	3S-LENFT-N11
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	Trading/LiquidityPairMetadata.sol#L225

Description

According to the standard solidity convention, internal function names should start with an underscore. This convention is used throughout the repo, except for function trait() in the **`LiquidityPairMetadata`** contract.

Recommendation

Change the function name to "_trait"

Status

Addressed here: leNFT/contracts@940843e and leNFT/contracts@4bbbeb7

3S-LENFT-N12

Throughout code: variables should be immutable

Id	3S-LENFT-N12
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	

Description

Throughout the code base, there are multiple contracts with variables set in the initialization and not allowed to change. These variables should be set to immutable to save gas, since they will be stored in the bytecode instead of storage (preventing all the SLOADs).

Recommendation

Change the following variables to immutable.

LendingMarket:

- **`_addressProvider`**

LoanCenter

- **`_addressProvider`**

- **`_defaultCollectionsRiskParameters.maxLTV`** and **`_defaultCollectionsRiskParameters.liquidationThreshold`** (they should also be set as independent variables and not as a struct, since they are never loaded together)

GaugeController

- **`_addressProvider`**

- **`_initialRewards`**

Trustus

- `INITIAL_CHAIN_ID`
- `INITIAL_DOMAIN_SEPARATOR`

Status

Addressed here: leNFT/contracts@3a3d49f , leNFT/contracts@3c7808e and leNFT/contracts@defafa3

3S-LENFT-N13

Throughout codebase: multiple comments fixes

Id	3S-LENFT-N13
Classification	None
Severity	None
Likelihood	
Category	Suggestion
Relevant Links	protocol/WETHGateway.sol#L41-L42 protocol/WETHGateway.sol#L49 protocol/WETHGateway.sol#L74-L80 protocol/VotingEscrow.sol#L39

Description

There are some typos or missing natspec comments throughout codebase

Suggestion

WETHGateway

- missing natspec comment in constructor

- typo in natspec comment

- missing natspec comment for variable **`genesisNFTid`**

VotingEscrow

- typo in natspec comment: "calimable" → "claimable"

Status

Addressed here: leNFT/contracts@9241d1d

3S-LENFT-N14

protocol/Gauges/LendingGauge.sol: Entire WorkingBalance struct is being loaded from storage just to access one field

Id	3S-LENFT-N14
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	Gauges/LendingGauge.sol#L272-L277 Gauges/LendingGauge.sol#L288

Description

The current implementation of the **`_checkpoint()`** function of the LendingGauge contract has the following code to load the WorkingBalance struct from storage:

```
DataTypes.WorkingBalance memory oldWorkingBalance;
if (_workingBalanceHistory[user].length > 0) {
    oldWorkingBalance = _workingBalanceHistory[user][
        _workingBalanceHistory[user].length - 1
    ];
}
```

Since the **`oldWorkingBalance`** variable is only used once to read the **`oldWorkingBalance.weight`**, this variable should be loaded instead of the entire struct.

Recommendation

Replace the above code with:

```
uint256 oldWorkingBalanceWeight;
if (_workingBalanceHistory[user].length > 0) {
    oldWorkingBalanceWeight = _workingBalanceHistory[user][
        _workingBalanceHistory[user].length - 1].weight;
```

```
}
```

And change variable **`oldWorkingBalance.weight`** to **`oldWorkingBalanceWeight`** on line 288

Status

Addressed here: leNFT/contracts@59f8f0d and leNFT/contracts@220687e

3S-LENFT-N15

Throughout codebase: rename variables to better represent their meaning

Id	3S-LENFT-N15
Classification	None
Severity	None
Likelihood	
Category	Suggestion
Relevant Links	Lending/LoanCenter.sol#L269 types/ConfigTypes.sol#L15

Description

Some variables should be renamed to better represent their use and meaning.

Recommendation

- LoanCenter: Lines 276 and 280, rename 'tokensPrice' to 'NFTsPrice' or 'CollateralPrice' to better represent its meaning ('tokens' is very generic)

- ConfigTypes, struct LendingPoolConfig, "auctioneerFee" → "auctioneerFeeRate", since the true actioner fee is this value times the loan debt

Status

Addressed here: leNFT/contracts@9113329

3S-LENFT-N16

protocol/VotingEscrow.sol: unnecessary modifiers

Id	3S-LENFT-N16
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	

Description

On the VotingEscrow contract, functions:

- **`increaseAmount()`**
- **`increaseUnlockTime()`**
- **`withdraw()`**
- claimRebates()

implement the modifiers **`lockExists(tokenId)`** and **`lockOwner(tokenId)`**, which check that the token exists (owner is not the zero address) and that the message sender is the token owner, respectively.

Here, since **`msg.sender`** will never be the zero address, the **`lockExists(tokenId)`** modifier is redundant.

Recommendation

On the functions listed above, remove the **`lockExists(tokenId)`** modifier

Status

Addressed here: leNFT/contracts@bb8708a

3S-LENFT-N17

protocol/Lending/LoanCenter.sol: missing natspec comments & variable naming

Id	3S-LENFT-N17
Classification	None
Severity	None
Likelihood	
Category	Suggestion
Relevant Links	Lending/LoanCenter.sol#L78-L95

Description

Natspec comments for `createLoan()` function is missing comment for **`owner`** argument

Recommendation

- Add comment describing **`owner`** argument

- Additionally, change argument name, since **`owner`** can be confused with the contract owner of the **`LoanCenter`** (since the contract is Ownable and has a callable function **`owner()`**)

Status

Addressed here: leNFT/contracts@db1971f

3S-LENFT-N18

protocol/VotingEscrow.sol: unnecessary struct storage load

Id	3S-LENFT-N18
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	protocol/VotingEscrow.sol#L443 protocol/VotingEscrow.sol#L447

Description

On the VotingEscrow contract, when creating a lock, line 443

`DataTypes.LockedBalance memory oldLocked = _lockedBalance[tokenId];` will always return a struct with 0's, so there is no need to load this struct from storage.

Similarly, in line 447, the struct that was just written to storage is being loaded.

Recommendation

- Replace line 443 with **`DataTypes.LockedBalance memory oldLocked = DataTypes.LockedBalance(0,0);`**

- Replace line 447 with **`_checkpoint(tokenId, oldLocked, DataTypes.LockedBalance(amount, roundedUnlockTime));`**

This would prevent 4 SLOADs, saving a considerable amount of gas

Status

Addressed here: leNFT/contracts@45a7cda and leNFT/contracts@871161d

3S-LENFT-N19

protocol/GenesisNFT.sol: remove always false condition

Id	3S-LENFT-N19
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	protocol/GenesisNFT.sol#L243-L245

Description

The genesis NFT contract, function **`_getCircleColor()`** has the following statement:

```
if (MAX_LOCKTIME == 0) {
    return "000000"; // return black
}
```

Since **`MAX_LOCKTIME`** is a constant equal to 180 days it can never be zero.

Recommendation

Remove this statement

Status

Addressed here: leNFT/contracts@1d247b6

3S-LENFT-N20

protocol/VotingEscrow.sol: duplicated requirement

Id	3S-LENFT-N20
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	protocol/VotingEscrow.sol#L323-L335

Description

The current implementation of the VotingEscrow has the following function:

```
function getLockHistoryPoint(uint256 tokenId,uint256 index) public view
returns (DataTypes.Point memory) {
    require(index < _lockHistory[tokenId].length, "VE:GLHP:INDEX_TOO_HIGH");
    return _lockHistory[tokenId][index];
}
```

Here, the `require(index < _lockHistory[tokenId].length, "VE:GLHP:INDEX_TOO_HIGH");` isn't necessary since solidity automatically adds this check by default when accessing an array element (at the moment this check is being run twice)

Recommendation

Remove line 333, `require(index < _lockHistory[tokenId].length, "VE:GLHP:INDEX_TOO_HIGH");`

Status

Addressed here: leNFT/contracts@9ff315b

3S-LENFT-N21

protocol/Lending/InterestRate.sol: cheaper to store the OptimalBorrowRate

Id	3S-LENFT-N21
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	Lending/InterestRate.sol#L116

Description

Calling function `getOptimalBorrowRate()` is expensive, since it requires 3 storage loads and some mathematical computations.

Recommendation

- It would be cheaper, gas-wise, to store the **`OptimalBorrowRate`** in the **`ConfigTypes.InterestRateConfig`** struct, in the **`_interestRateConfigs`** mapping and just load it from storage whenever needed, since the value of this variable is only dependant on other config values.

- To make sure the formula is correct, the **`OptimalBorrowRate`** could be calculated inside function **`setInterestRateConfig()`**, this way it would only be executed once per token

Status

Addressed here: leNFT/contracts@188036a, leNFT/contracts@ca9aeca, leNFT/contracts@3339a7c and leNFT/contracts@56f766c

3S-LENFT-N22

Trading/TradingPool.sol: nonReentrant modifier being called many times inside a function

Id	3S-LENFT-N22
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	

Description

The current implementation of the **`removeLiquidityBatch(uint256[] lpIds)`** function calls the function **`removeLiquidity(uint256 lpId)`** for every LP ID in the array. Since function **`removeLiquidity(uint256 lpId)`** is public and reentrancy protected, removing liquidity by batch will result in switching the mutex variable (of the nonReentrant modifier) on and off multiple times, leading to gas waste.

Recommendation

To avoid constantly setting the nonReentrant mutex on and off, consider writing the removeLiquidity logic in a private function, without the nonReentrant modifier, and calling this function from the external functions **`removeLiquidityBatch`** and **`removeLiquidity`**, (which would be reentrancy protected)

Status

Addressed here: leNFT/contracts@ed7c261 and leNFT/contracts@77e1657

3S-LENFT-N23

Trading/SwapRouter.sol: Replace line to reduce gas and improve readability

Id	3S-LENFT-N23
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	

Description

The current implementation of the SwapRouter, function swap() has the following statement:

```
// If the price difference + sell price is greater than the buy price,
return the difference to the user
if (sellPrice + priceDiff > buyPrice) {
    IERC20(sellPoolToken).safeTransfer(
        msg.sender,
        sellPrice + priceDiff - buyPrice
    );
    change = sellPrice + priceDiff - buyPrice;
}
```

Here, the variable **`change`**, i.e., **`sellPrice + priceDiff - buyPrice`** is being computed twice.

Recommendation

Change the statement to the following to improve readability and improve gas usage:

```
// If the price difference + sell price is greater than the buy price,
return the difference to the user
if (sellPrice + priceDiff > buyPrice) {
```

```
    change = sellPrice + priceDiff - buyPrice;
    IERC20(sellPoolToken).safeTransfer(
        msg.sender,
        change
    );
}
```

Status

Addressed here: leNFT/contracts@03fda52

3S-LENFT-N24

protocol/Lending/LoanCenter.sol: functions repayLoan() and liquidateLoan() share most of the code

Id	3S-LENFT-N24
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	Lending/LoanCenter.sol#L130-L185

Description

In the LoanCenter, functions repayLoan() and liquidateLoan() have the exact same code (except for the first line that sets the loanState).

Recommendation

Knowing this, an auxiliary internal function should be created to close a loan, i.e.:

- Delete the mapping from NFT to loan ID

- Remove loan from user active loans

Creating this internal function would substantially reduce the LoanCenter contract size and improve code readability.

Status

Addressed here: leNFT/contracts@03fda52 and leNFT/contracts@5e8edb3

3S-LENFT-N25

Bribes.sol: Rework loop to save gas

Id	3S-LENFT-N25
Classification	None
Severity	None
Likelihood	
Category	Optimization
Relevant Links	

Description

On the claim() function of the Bribes contract, the following loop can be simplified (saving gas in storage reads and writes):

```
uint256 epoch;
for (uint i = 0; i < 50; i++) {
    // Break if we're at the current epoch or higher
    epoch = _voteNextClaimableEpoch[token][gauge][tokenId];
    if (epoch > currentEpoch) {
        break;
    }
    (...)
    // Increment epoch
    _voteNextClaimableEpoch[token][gauge][tokenId]++;
}
```

Recommendation

Rewrite the loop as:

```
uint256 epoch = _voteNextClaimableEpoch[token][gauge][tokenId];
for (uint i = 0; i < 50 && epoch <= currentEpoch; i++) {
    (...)
    // Increment epoch
    epoch++;
```

```
}
_voteNextClaimableEpoch[token][gauge][tokenId] = epoch;
```

Note: The loop could even be further optimized by incrementing `i` without checking for overflow:

```
uint256 epoch = _voteNextClaimableEpoch[token][gauge][tokenId];
for (uint i = 0; i < 50 && epoch <= currentEpoch;) {
    (...)
    // Increment epoch
    epoch++;
    unchecked {
       ++i;
    }
}
_voteNextClaimableEpoch[token][gauge][tokenId] = epoch;
```

Status

Addressed here: leNFT/contracts@2198a24 and leNFT/contracts@5f0c82b

3S-LENFT-N26

GenesisNFT.sol: use payable(address(this)) instead of payable(this)

Id	3S-LENFT-N26
Classification	None
Severity	None
Likelihood	
Category	Suggestion
Relevant Links	

Description

The current implementation of the GenesisNFT.sol contract has the following receive function:

```
// Function to receive Ether
receive() external payable {
    revert("G:RECEIVE_NOT_ALLOWED");
}
```

This function always reverts, and only exists because the exitPool() function of the balancer vaults require an address payable as argument, and the current GenesisNFT.sol contract uses the **`payable(this)`** to retrieve the payable address.

Recommendation

Casting the contract to an address first, and then to a payable address (i.e. **`payable(address(this))`**) would allow the removal of this function, maintaining the exact same behavior (of reverting on ether received).

Status

Addressed here: leNFT/contracts@7817040