BAIL security

Terminal Finance DEX

FINAL REPORT

June ‘2025

Disclaimer:

Security assessment projects are time-boxed and often reliant on information that may be provided by a client, its affiliates, or its partners. As a result, the findings documented in this report should not be considered a comprehensive list of security issues, flaws, or defects in the target system or codebase.

The content of this assessment is not an investment. The information provided in this report is for general informational purposes only and is not intended as investment, legal, financial, regulatory, or tax advice. The report is based on a limited review of the materials and documentation provided at the time of the audit, and the audit results may not be complete or identify all possible vulnerabilities or issues. The audit is provided on an "as-is," "where-is," and "as-available" basis, and the use of blockchain technology is subject to unknown risks and flaws.

The audit does not constitute an endorsement of any particular project or team, and we make no warranties, expressed or implied, regarding the accuracy, reliability, completeness, or availability of the report, its content, or any associated services or products. We disclaim all warranties, including the implied warranties of merchantability, fitness for a particular purpose, and non-infringement.

We assume no responsibility for any product or service advertised or offered by a third party through the report, any open-source or third-party software, code, libraries, materials, or information linked to, called by, referenced by, or accessible through the report, its content, and the related services and products. We will not be liable for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract.

The contract owner is responsible for making their own decisions based on the audit report and should seek additional professional advice if needed. The audit firm or individual assumes no liability for any loss or damages incurred as a result of the use or reliance on the audit report or the smart contract. The contract owner agrees to indemnify and hold harmless the audit firm or individual from any and all claims, damages, expenses, or liabilities arising from the use or reliance on the audit report or the smart contract.

By engaging in a smart contract audit, the contract owner acknowledges and agrees to the terms of this disclaimer.

1. Project Details

Important:
Please ensure that the deployed contract matches the source-code of the last commit hash.

Project	TerminalFi - Scope 1
Website	terminal.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/terminal-finance/contracts/tree/7965ab9b37def79f386c71cc85c407fd42f7e06a
Resolution 1	https://github.com/terminal-finance/contracts/tree/3ef5a12896cecb2f3298a7f1fd889a5e566a2db4/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)	Failed resolution
High	8	6	1	1	
Medium	17	9	3	5	
Low	24	4		20	
Informational	27	6		21	
Governance	2			2	
Total	78	25	4	49	

2.1 Detection Definitions

Severity	Description
High	The problem poses a significant threat to the confidentiality of a considerable number of users' sensitive data. It also has the potential to cause severe damage to the client's reputation or result in substantial financial losses for both the client and the affected users.
Medium	While medium level vulnerabilities may not be easy to exploit, they can still have a major impact on the execution of a smart contract. For instance, they may allow public access to critical functions, which could lead to serious consequences.
Low	Poses a very low-level risk to the project or users. Nevertheless the issue should be fixed immediately
Informational	Effects are small and do not post an immediate danger to the project or users
Governance	Governance privileges which can directly result in a loss of funds or other potential undesired behavior

3. Detection

Core

All core contracts are forked from UniswapV3: https://github.com/Uniswap/v3-core/tree/c5ccf4d28a73fde90f0bb9ea3fd299d7d2bcdf83/contracts

While most contracts are identical, some contracts are modified. Only the modified contracts are part of the scope and will be listed below. A differential audit will be conducted and any potential issue in the original UniswapV3 scope will be disregarded.

TerminalPoolFactory

The TerminalPoolFactory is forked from the UniswapV3 factory contract and has introduced a new deployment process which is based on the fact that pools are deployed during the createPool function within the Voter contract. The diff can be found here: https://www.diffchecker.com/wnvczGZn/

The main difference therefore is the prevention of permissionless pool deployments.

Appendix: Deployment Process

The Voter contract offers a createPool function which allows the governor to create a pool for whitelisted tokens. This will then first invoke the createPool function on the factory to deploy the TerminalPool, followed by the deployment of the corresponding gauge and all Vaults as well as the corresponding setting.

Core Invariants (based on changes):

INV 1: Pools must only be created via the Voter contract

Privileged Functions

- createPool
- setVoter

No issues found.

TerminalPoolDeployer

The TerminalPoolDeployer is forked from the UniswapV3 factory contract but heavily modified. The diff can be found here: https://www.diffchecker.com/UBJ9zDgl/

It first alters the contract storage for the corresponding deployment parameters, then deploys the TerminalPool via a create2 call, fetches the storage within the TerminalPool constructor and then resets storage once the TerminalPool has been deployed.

Privileged Functions

- none

Issue_01	Permissioned pool deployment prevents launchpads being built on top
Severity	**Low**
Description	In the current web3 environment, a lot of launchpads are being built on top of UniswapV3. These launchpads often create a pair and add liquidity in the same transaction. Due to the permissioned nature of the pool deployment which is only possible by governance, such a launchpad use case is completely ruled out. This can result in intrusive limitations of possibilities for the overall DEX system. Furthermore, there is no application of the know UniswapV3 createAndInitializePoolIfNecessary pattern which allows users to change the price post deployment.
Recommendations	Consider if it is desired to support such launchpads. In such a scenario the deployment flow must be refactored and whitelisted partners should be allowed to deploy pools.
Comments / Resolution	Acknowledged.

TerminalPool

The TerminalPool contract is forked from the UniswapV3Pool contract and implements a **fundamental rehaul of the fee distribution** as well as an **introduction of redeemable tokens**. The diff can be found here: https://www.diffchecker.com/ClzWN9jO/

Originally, UniswapV3 distributes swap and flash fees among the current active liquidity and does not support rebase tokens. The Terminal team refactored the UniswapV3Pool contract with the goal to be able to support their native RedeemableERC20 tokens which is a rebasing token with an ever increasing supply.

Furthermore, there are two ways to provide liquidity:

a) Unstaked Liquidity: This is the standard known UniswapV3-like liquidity provision which receives a part of the swap and flash fees. Additionally, a part of the rebasing yield is distributed among the active unstaked liquidity as well. The rest of the swap/flash/rebase fee is allocated in a specific GaugeRevenue struct which is claimed at the beginning of each epoch and allocated towards VE voters. A more detailed explanation can be found in the appendix below.

b) Staked Liquidity: The Terminal team has implemented an additional parameter in the Position struct determination which is the isStaked parameter. Users can provide staked liquidity, which technically does not differ from the normal liquidity and is also used for swaps and everything else. However, instead of swap/flash/rebase fees, the stakedLiquidity receives rewards in form of the TERM token which is provided by the Voter contract upon each epoch flip. If, for example 100_000e18 TERM tokens are minted by the Voter upon an epoch flip and the pool has a weight of 10% (which is based on votes in the previous epoch). 10_000e18 TERM tokens will be distributed evenly over the course of the newly started epoch among the stakedLiquidity.

Appendix: Reward Distribution

As above mentioned, the TerminalPool contract allows for staked and unstaked liquidity providing. Unstaked liquidity receives rewards in form of TERM tokens which is accounted for via the rewardGrowthInsideLastX128 variable. The mechanics are completely identical to the normal fee distribution, however, the rate is determined via the Gauge.getRewardGrowth function which drips the TERM amount received during the beginning of the epoch between [epochEndTimestamp; notificationTimestamp] by calculating the corresponding rewardRate:

Rate calculation upon epoch flip:

> rewardRate = rewards / (epochEndTimestamp - notificationTimestamp)

Calculation of rewards since last update:

> reward = rewardRate * timeElapsed

The reward is then divided by the stakedLiquidity in an effort to account for each liquidity unit (scaled by 128):

> rewardGrowthDeltaX128 = reward * (2**128) / stakedLiquidity

Afterwards, the delta is added to the global variable:

> rewardGrowthGlobalX128 += rewardGrowthDeltaX128

A special feature is the rollover variable which gets incremented whenever the _getRewardGrowth function is called with a stakedBalance of zero. This can be the case whenever notifyRewardAmount/Boost is called or whenever the pool is in a state with no active stakedLiquidity.

Appendix: Fee Distribution

The fee distribution is highly aligned with the standard UniswapV3 fee distribution. Fees are accumulated via swaps, flashloans and via rebasing rewards which are dripped time-based over 24 hours based on the PendingYield library (as explained in Appendix: Rebase Distribution). All of these fees are aggregated in token0/1 and accounted for via the revenueGrowthGlobal0X128/1x128 variable.

However, contrary to UniswapV3, not the full amount of these fees is distributed to the (unstaked) liquidity. Instead a part is collected for the Gauge which is then distributed via the Gauge towards the FeeVault and YieldVault. The exact amount is calculated upon each swap/flash/drip via the _splitRevenue function is based on the proportion of the stakedLiquidity to the overall liquidity as well as the unstakedFeeTax and unstakedYieldTax.

> Calculate gaugeRevenueDelta

> gaugeRevenueDelta = amount * liquidity / stakedLiquidity

> Increase gaugeRevenueDelta by potential tax
> (amount - gaugeRevenueDelta) / unstakedTax
> gaugeRevenueDelta += amount

> Calculate leftover amount for liquidity providers
> growthGlobalDeltaX128 = (amount - gaugeRevenueDelta) * (2**128) / (liquidity - stakedLiquidity)

Appendix: Rebase Distribution

As already mentioned, the contract is developed in such a manner that it supports the protocol owned RedeemableERC20 wrapper contract for tokens which increase in price. Instead of keeping the price increase for these tokens, the supply will be increased. This supply increase will be reflected via the _updateYieldGrowth function which leverages the PendingYield library to drip these rebasing fees over over 24 hours. The dripping mechanism is completely outsourced to this library and will be explained in the description of the PendingYield contract section.

This yield will then be split into a part which goes towards the Gauge and corresponding YieldVault for voting rewards (gaugeRevenue.yield0/1) and a part which will be distributed to all liquidity suppliers for unstaked liquidity in the active range (revenueGrowthGlobalX128). The yield distribution calculation towards liquidity providers and the gauge follows the exact same calculation as described above (_splitRevenue function).

Core Invariants (based on changes):

INV 1: Nominal liquidity must always be >= stakedLiquidity

INV 2: Within _splitRevenue, feeGrowth + gaugeFee must be equal to feeAmount

INV 3: Return values from _splitRevenue must increase revenueGrowthGlobal0/1 and gaugeRevenue.fee0/1 OR gaugeRevenue.yield0/1

INV 4: Term rewards must be distributed among the stakedLiquidity

INV 5: _updateRewardGrowth must be called at the beginning of the swap and _updatePosition functions.

INV 5: _updateYieldGrowth must be called at the beginning of the swap and _updatePosition functions.

INV 6: gaugeRevenue.fee0/1 and gaugeRevenue.yield0/1 must be fully distributed to the Gauge upon collectGaugeRevenue

INV 7: stakedLiquidityNet must follow the exact same mechanics as liquidityNet

INV 8: stakedLiquidity must follow the exact same mechanics as liquidity

Privileged Functions

- setUnstakedTax
- collectGaugeRevenue

Issue_02	Rebase rewards can be stolen via swap
Severity	**High**
Description	In the first iteration, the previous auditor found an issue which allowed flash-thefting of rebase rewards via simple flashloan and price update. This has been fixed such that a transfer cannot be executed by the same initiator as a price update (in the same block). However, there is still a possibility to flash-theft rebase rewards by leveraging a very specific attack path. Illustrated: a) Alice creates a contract which serves as recipient of the swap. This contract has an access control mechanism with an execute function which can be called by Alice A and Alice B (not even necessary but simplifies the PoC) b) Alice A executes a swap to buy rebaseX with tokenY which is received by the contract. In the same transaction, Alice A calls the price update function. c) Immediately followed by this transaction, Alice B calls the execute function on the contract which allows her to swap back rebaseX and buy back tokenY from the pair. This is possible because Alice B is now the tx.origin, if called by Alice A, it would revert due to the safeguard which was implemented as fix. Using that technique, Alice was able to flash-theft due rebase rewards from RedeemableERC20X while only paying for the swap fee. Moreover, this attack is technically feasible by MEV bots in the same known manner as sandwich attacks using standard strategies such as special gas values, private mempool, etc.

	It **does not require multiple blocks** to be executed and does not require bribing block producers, thus it is technically feasible without any issues and is considered a high-risk issue.
Recommendations	Consider implementing an incentive mechanism in the price oracle which incentivizes users to update the price more often. While this will only partially resolve that issue, it makes the impact almost negligible.
Comments / Resolution	Acknowledged. The client intends to frequently update the price oracle such that yield < swap fee.

Issue_03	Change of unstakedYieldTax will have hindsight impact
Severity	**Medium**
Description	The unstakedYieldTax value determines the share which is additionally allocated to the Gauge from the rebasing yield. Contrary to flash and swap fees, this mechanism is time-based. Due to this fact, any change in unstakedYieldTax will impact the rebase yield from [pendingYield.lastDrip; block.timestamp].
Recommendations	Consider calling _updateYieldGrowth at the beginning of the setUnstakedTax function.
Comments / Resolution	Resolved by following the recommendation.

Issue_04	Rebase yield sharing can always be bypassed
Severity	**Medium**
Description	As explained in the appendix, the rebase yield is always shared with the current active liquidity. This can be trivially bypassed by a user via withdrawing liquidity, updating the price and re-adding liquidity in the next block. That way the user has gained his full rebase rewards while not losing out from more than 1 block of the pool's rebase rewards (since these are dripped out only).
Recommendations	Consider implementing an incentive mechanism in the price oracle which incentivizes users to update the price more often. While this will only partially resolve that issue, it makes the impact almost negligible.
Comments / Resolution	Acknowledged. The clients intends to frequently update the price oracle.

Issue_05	Incorrect return value for growthGlobals
Severity	**Medium**
Description	The growthGlobals function returns the current state of rewards: *return (* *revenueGrowthGlobal0X128,* *revenueGrowthGlobal1X128,* *rewardGrowthGlobalX128* *);* This is inaccurate since rewardGrowthGlobal is time-based and will thus return an old value, not incorporating rewards from [lastUpdateTime; block.timestamp]. This will be an inherent issue for protocols which build on top of Terminal and rely on the correctness of this function.
Recommendations	Consider if it makes sense to invoke the _updateRewardGrowth function. This would however remove the view-only nature of the function. Alternatively, a comment to the function can be added which notifies developers that this state is unsafe and the pool must be updated beforehand.
Comments / Resolution	Resolved by following recommendation to add a comment above the function (in ITerminalPoolState.sol).

Issue_06	Nominal underlyingToken rewards may be lost due to price update before collect
Severity	**Low**
Description	The collect function only stores the RedeemableERC20 rewards which present a specific amount of the underlying token. This is depending on the price, the higher the price, the smaller the underlying token amount (for the same amount of RedeemableERC20). Thus naturally if the price is updated before a user collects, the user will receive the same amount of RedeemableERC20 tokens which are however worth less underlying tokens.
Recommendations	Consider implementing an incentive mechanism in the price oracle which incentivizes users to update the price more often. This will prevent large price updates and thus the nominal token amount will not suddenly increase by a large margin.
Comments / Resolution	Acknowledged. The clients intendes to frequently update the price oracle.

Issue_07	Price update before collect results in user receiving less nominal underlying token
Severity	**Low**
Description	In the scenario where one or both tokens are the RedeemableERC20 token, fees and yield are accrued in the said token and can then be collected via the collect function. If the price of the RedeemableERC20 is 1e8 while the user has 100e18 idle RedeemableERC20 tokens, this means the user effectively owns 100e18 of the underlying token (if he would claim). However, if a price update is executed before the claim and the price increases to 1.1e8, the user effectively lost tokens and now only owns 90.09e18 underlying tokens. This issue is inherent based on the design.
Recommendations	Consider implementing an incentive mechanism in the price oracle which incentivizes users to update the price more often. Moreover, regular claiming should be conducted to avoid donating a part of the owned yield/fee to the pair.
Comments / Resolution	Acknowledged.

Issue_08	Storage variables can be marked immutable
Severity	**Informational**
Description	Several storage variables within the TerminalPool, including factory, term, token0, token1, fee, redeemable0, redeemable1, tickSpacing, and maxLiquidityPerTick, are set once in the constructor and remain unchanged afterward. As an optimization, these variables can be declared as immutable instead.
Recommendations	Consider marking the mentioned variables as immutable.
Comments / Resolution	Acknowledged. The client indicated that marking these variables as immutable bloats the contract size past 24.5kB.

Issue_09	Lack of support for transfer-tax tokens
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using such tokens.
Comments / Resolution	Acknowledged.

Issue_10	Codebase is not compatible with tokens that have a super high supply
Severity	**Informational**
Description	Similar as with Uniswap, multiple spots in the codebase are using an unsafe casting to uint128 with a comment that overflow is accepted. This obvious scenario can for example result in a loss of rewards.
Recommendations	Consider conducting due diligence on which tokens will be whitelisted for the pair creation.
Comments / Resolution	Acknowledged.

Issue_11	unstakedTax is initially zero
Severity	**Informational**
Description	Currently, both unstaked taxes are initially zero and will only be set once the setUnstakedTax function is called. While this may be a design choice, it can result in these fees being forgotten to set for a certain time.
Recommendations	Consider if it makes sense to set these fees upon deployment to a default value.
Comments / Resolution	Acknowledged.

Issue_12	Token collection does not updated state
Severity	**Informational**
Description	The collect function allows for withdrawing rewards as well as burned tokens, based on the aggregated amounts in owed.amount0/owed.amount1/owed.reward. It has to be noted that all three values can be time-dependent, e.g. some rewards may still be dripped or claimed from the gauge. This is not only related to time-based rewards but also to the fact that the innerGrowth for all fees is not updated. The collect function does not update the pool and position state to reflect these unaggregated rewards. This is however not a big issue since the user can simply update the position before via a zero burn (which is anyways done by the NFPM).
Recommendations	Since this issue is also present in UniswapV3, we recommend acknowledging it.
Comments / Resolution	Acknowledged.

CollectAmounts

The CollectAmounts library is a simple representation of the Info struct which is used for the collect function and the PositionInfo.info struct:

> amount0
> amount1
> rewards

Privileged Functions

- None

No issues found.

Oracle

The Oracle library is a simple helper library which is responsible for the UniswapV3 TWAP oracle. It is almost identical to the original UniswapV3 Oracle with the only change being a visibility adjustment in the observe and _*observeSingle functions in an effort to reduce the contract size of the TerminalPool contract.

Privileged Functions

- None

No issues found.

Position

The Position library is forked from the Position library in UniswapV3 and implements the new isStaked trait for positions and incorporates the new fee structure. The diff can be found here: https://www.diffchecker.com/vL5g1jZF/

Core Invariants (based on changes):

INV 1: Instead of tokensOwed1 and tokensOwed 2, each position has a corresponding CollectAmounts.info struct which represents owed tokens and reward

INV 2: Positions must differ based on isStaked trait

INV 3: The update function must have separate conditions whether a position isStaked or not

INV 4: rewardGrowthInsideLastX128 must be set to the latest updated rewardGrowthInside during update (isStaked = true)

INV 5: revenueGrowthInsideLast0/1X128 must be set to the latest updated revenueGrowthInside0/1X128 during update (isStaked = false)

INV 6: self.liquidity must only be changed after fee aggregation

Privileged Functions

- None

No issues found.

Tick

The Tick library is forked from the Tick library in UniswapV3 and implements additional variables for the Info struct of a tick to represent the stakedLiquidityNet as well as the new fee variables. Moreover it implements a getRewardGrowthInside function which 1:1 mimics the getFeeGrowthInside mechanism from UniswapV3. Whenever a tick is updated or crossed it will handle the newly introduced fees as well as the stakedLiquidityNet. The diff can be found here: https://www.diffchecker.com/20OpFNKe/

Core Invariants (based on changes):

INV 1: getRewardGrowthInside and getRevenueGrowthInside must exactly mimic the getFeeGrowthInside algorithm from UniswapV3

INV 2: stakedLiquidityNet must mimic the same mechanics as liquidityNet

Privileged Functions

- none

No issues found.

PendingYield

The PendingYield library is utilized during the _updateYieldGrowth function and drips yield rewards in an interval of 24 hours towards the active liquidity.

Appendix: Drip Mechanism

Whenever a price update has happened this means yield is claimable via the RedeemERC20.redeem function. To counter flash-theft attacks, this yield is not distributed immediately but linearly over the course of 24 hours, using the following formula:

> yield = (amount / (endTime - lastDrip) * (blockTimestamp - lastDripTime)

Contrary to most linear vesting mechanisms, the amount is always decreased by the dripped amount after each call, which aligns with the above math formula and ensures a linear vesting over the course of the DRIP_INTERVAL, which is 24 hours

Core Invariants (based on changes):

INV 1: Whenever redeem returns a non-zero amount, the interval must start from the current block.timestamp

INV 2: Whenever endTime is passed, the full leftover amount must be distributed as yield

Privileged Functions

- none

Issue_13	Division before multiplication results in truncation
Severity	**Informational**
Description	Generally speaking in solidity, any division before multiplication is prone to truncation. The same applies to the yield calculation in the first condition: *yield = (self.amount / (self.endTime - self.lastDrip)) * (time - self.lastDrip);* Since the division is only timestamp based, this issue can be safely acknowledged.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_14	Unsafe casting during drip
Severity	**Informational**
Description	The amount which is returned by the RedeemableERC20.redeem function is casted to uint128 which can result in an overflow and thus a loss of rewards. This is only rated as an informational severity due to the fact that multiple other spots in the core explicitly assume that the supply will never exceed uint128.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Periphery

All core contracts are forked from UniswapV3: https://github.com/Uniswap/v3-periphery/tree/0682387198a24c7cd63566a2c58398533860a5d1

While most contracts are identical, some contracts are modified. Only the modified contracts are part of the scope and will be listed below. A differential audit will be conducted and any potential issue in the original UniswapV3 scope will be disregarded.

LiquidityManagement

The LiquidityManagement library is forked from the LiquidityManagement library in UniswapV3. The only difference is the fact that the poolInitCodeHash is not hardcoded within the CallbackValidation library and thus provided as parameter as well as the implementation of a payer and isStaked parameter for liquidity addition. The diff can be found here: https://www.diffchecker.com/UVEzvqkg/

Privileged Functions

- none

No issues found.

PeripheryImmutableState

The PeripheryImmutableState library is forked from the PeripheryImmutableState library in UniswapV3. The only addition is the poolInitCodeHash which is stored during the contract deployment. In UniswapV3 it is hardcoded into the CallbackValidation library.

Privileged Functions

- none

No issues found.

PeripheryPayments

The PeripheryPayments library is forked from the PeripheryPayments library in UniswapV3. It exposes two new functions:

a) unwrapRedeemable: This function is directly callable by a user and allows to unwrap the RedeemableERC20 token with the recipient of the underlyingToken being an arbitrary address

b) wrapRedeemable: This function is being triggered inside the pay function and is responsible for transferring the corresponding underlying amount for the desired RedeemableERC20 amount from the user to this contract, which is then wrapped in form of minted RedeemableERC20 tokens to the recipient

The diff can be found here: https://www.diffchecker.com/acTve8Be/

Privileged Functions

- none

Issue_15	Lack of token validation within unwrapRedeemable
Severity	Medium
Description	The unwrapRedeemable function is used to clean up leftover RedeemableERC20 tokens held by contracts like the NonfungiblePositionManager or SwapRouter. It takes a token and a recipient address, then calls: *token.burn(recipient, token.balanceOf(address(this)))* This assumes the token uses RedeemableERC20 logic, where burn(address,uint256) burns tokens from the msg.sender and sends the proceeds to the recipient. However, some tokens, like DAI, define burn(address,uint256) with different behavior. DAI treats the address as the source of tokens and uses allowances if the caller isn't the same as the address. So, since the NonfungiblePositionManager and SwapRouter contracts do hold allowances, an attacker could call unwrapRedeemable with DAI and a victim's address to potentially burn the victim's tokens. Note that in DAI's case, this specific attack fails since its burn function doesn't return a uint256, which would cause the logic to revert. However, it is likely that such a token exists and may be used within this scope. This issue is only rated as medium severity since we could not find such a token at this moment. However, it must be fixed under all circumstances.
Recommendations	Consider adding a requirement that RedeemableChecker.isRedeemable(token) is true in unwrapRedeemable. This would ensure that the function can only be used with tokens that specify they follow the RedeemableERC20.burn() logic.
Comments / Resolution	Resolved by following the recommendation.

Issue_16	No functionality to transfer actual RedeemableERC20 token in
Severity	**Low**
Description	The pay function exposes a special case in the scenario where the token is the RedeemableERC20 token: *} else if (RedeemableChecker.isRedeemable(token)) {* *// unwrap redeemable token as payment* *wrapRedeemable(token, payer, recipient, value);* This condition specifically transfers in the underlying token and wraps it. Even if a user owns the RedeemableERC20 token, it is not supported to transfer it in the normal way.
Recommendations	Consider if it makes sense to implement functionality for transferring the owned RedeemableERC20 token in. Alternatively, this issue can be acknowledged.
Comments / Resolution	Acknowledged.

NonfungiblePositionManager

The NonfungiblePositionManager contract is forked from the NonfungiblePositionManager contract in UniswapV3. Multiple changes and additions were made:

a) Both positions (isStaked = true; isStaked = false) can be minted

b) It is now allowed to use address(this) as payer for the mint function

c) All liquidity adjusting functions are modified in an effort to handle the new fee/reward model

d) The collect function allows for additional claiming of TERM rewards

The diff can be found here: https://www.diffchecker.com/ocUDwrC8/

Core Invariants:

INV 1: updateGrowth must be called during increaseLiquidity and decreaseLiquidity with the pre change liquidity

INV 2: revenueGrowthInside0/1LastX128 and rewardGrowthInsideLastX128 must be updated before updateGrowth is called

INV 3: updateGrowth must always write fee to storage based on whether a position isStaked or not.

Privileged Functions

- none

Issue_17	Incorrect amount0Desired parameter in case of isPayerSelf = true can result in loss of funds
Severity	**Medium**

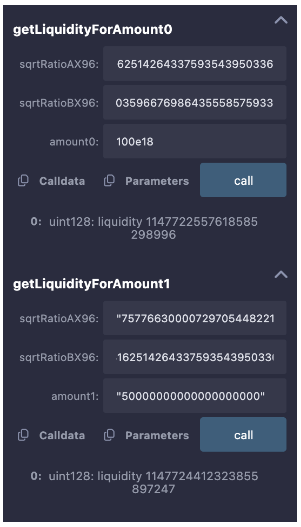
Description

Currently, an incorrect balanceOf check within the mint function is used.

If for example a user wants to add liquidity with the following setup:

sqrtPriceLow = 75776630000729705448221412452
currentPrice = 79228162514264337593543950336
upperPrice = 86790103596676986435558575933
amount0 = 100e18
amount1 = 50e18

yielding around 1147.72e18 in liquidity:



```
getLiquidityForAmount0
sqrtRatioAX96: 62514264337593543950336
sqrtRatioBX96: 03596676986435558575933
amount0: 100e18
Calldata   Parameters   call
0: uint128: liquidity 1147722557618585
298996

getLiquidityForAmount1
sqrtRatioAX96: "75776630000729705448221
sqrtRatioBX96: 1625142643375935439503З
amount1: "50000000000000000000"
Calldata   Parameters   call
0: uint128: liquidity 1147724412323855
897247
```

Furthermore, amount0Min and amount1Min are zero.

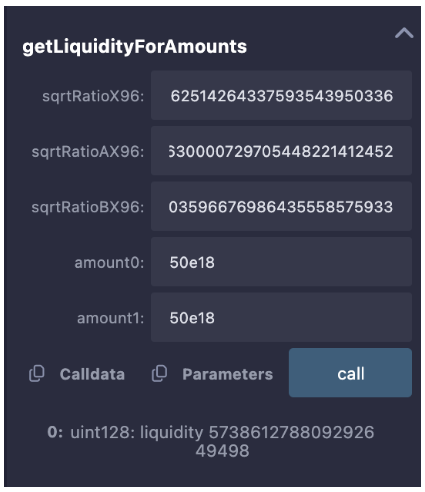
This would perfectly work out for the presented amounts. However, due to the blunder within the amount0Desired parameter, it will now use 50e18 for amount0Desired instead of 100e18.

The result will now be:

amount0Desired = 50e18
amount1Desired = 50e18

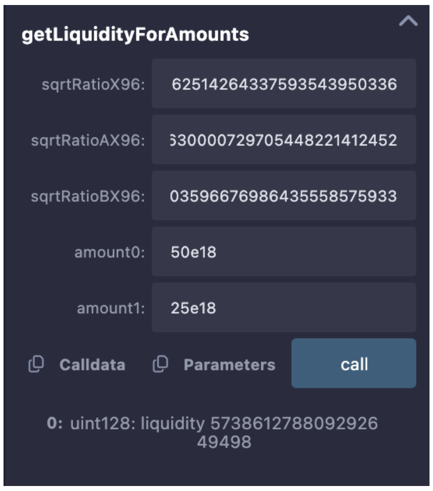
Due to the way how the core protocol works, it will now only add 50e18 of token0 and 25e18 of token1 (due to the range) which yields a liquidity of 573,86e18

One can see here, how 25e18 token1 and 50e18 token1 yield the same liquidity amount, which means that only 25e18 is taken:



```
getLiquidityForAmounts
  sqrtRatioX96:  6251426433759354395O336
 sqrtRatioAX96:  30000729705448221412452
 sqrtRatioBX96:  0359667698643555857593З
       amount0:  50e18
       amount1:  50e18
Calldata   Parameters   call

0: uint128: liquidity 573861278809292649498
```



```
getLiquidityForAmounts
  sqrtRatioX96:  62514264337593543950336
 sqrtRatioAX96:  30000729705448221412452
 sqrtRatioBX96:  03596676986435558575933
       amount0:  50e18
       amount1:  25e18
Calldata   Parameters   call

0: uint128: liquidity 573861278809292649498
```

The leftover amount basically remains in the NFPM and can be skimmed by subsequent users.

This issue has only been rated as medium severity instead of high severity since the user must provide loose slippage parameters. In

	case where correct slippage parameters are provided, the call will revert which effectively renders the isSelfPayer feature completely unusable (unless a user adds liquidity geometrically around the price of 1, which means the same amount of tokenX and tokenY is being taken)
Recommendations	Consider simply using the correct token for the amount0Desired parameter in case isSelfPayer = true.
Comments / Resolution	Resolved by following the recommendation.

Issue_18	Price update after liquidity supply will result in loss for provider
Severity	**Low**
Description	Users can provide liquidity to pools which consist of the RedeemableERC20 token. In the scenario where a user supplies liquidity with an unupdated price and another user updates the price immediately after liquidity has been supplied, the yield is then shared with the TerminalPool instead given to the liquidity supplier. This issue is inherent based on the design.
Recommendations	Consider implementing an incentive mechanism in the price oracle which incentivizes users to update the price more often.
Comments / Resolution	Acknowledged. The client intends to frequently update the price oracle.

Issue_19	Burn can be griefed
Severity	**Informational**
Description	The burn function requires the following condition to be true: *require(position.liquidity == 0 && position.owed.amount0 == 0 && position.owed.amount1 == 0 && position.owed.reward == 0, 'Not cleared');* This can be griefed due to the fact that the addLiquidity function is permissionless.
Recommendations	SInce this issue is present in UniswapV3 as well, we recommend acknowledging it.
Comments / Resolution	Acknowledged.

NonfungibleTokenPositionDescriptor

The NonfungibleTokenPositionDescriptor contract is forked from the NonfungibleTokenPositionDescriptor contract in UniswapV3. Only minor changes with regards to the new Position struct within the NFPM have been made.

The diff can be found here: https://www.diffchecker.com/E4dqGYOs/

Privileged Functions

- none

No issues found.

SwapRouter

The SwapRouter contract is forked from the SwapRouter contract in UniswapV3. It was solely adjusted to incorporate the fact that the poolInitCodeHash is not hardcoded within the CallbackValidation library but instead determined during contract deployment.

The diff can be found here: https://www.diffchecker.com/oCGSeLlu/

Privileged Functions

- none

Issue_20	Governance Privilege: Deployment with wrong parameters
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. It is possible to deploy the SwapRouter with a wrong factory and initCodeHash such that the callback can be abused to steal approvals. The same issue is apparent within the NFPM as well but will not be raised separately.
Recommendations	Consider ensuring that deployment parameters are always correct.
Comments / Resolution	Acknowledged.

CallbackValidation

The CallbackValidation library is forked from the CallbackValidation library in UniswapV3. It was solely adjusted to incorporate the fact that the poolInitCodeHash is not hardcoded.

The diff can be found here: https://www.diffchecker.com/reazeQNM/

Privileged Functions

- none

No issues found.

PositionKey

The PositionKey library is forked from the PositionKey library in UniswapV3. It was solely adjusted to incorporate the isStaked trait.

Privileged Functions

- none

No issues found.

PositionValue

The PositionValue library is forked from the PositionValue library in UniswapV3. It was adjusted in an effort to support the new fee/revenue mechanism and the isStaked trait.

Core Invariants:

INV 1: TERM rewards must only be incorporated for positions with the isStaked trait

INV 2: fee/yield revenue must only be incorporated for positions with the isStaked = false trait

Privileged Functions

- none

Issue_21	Insufficient differentiation between staked and unstaked positions
Severity	**High**
Description	Staked positions only accrue TERM rewards while unstaked positions only accrue fee/yield. There is currently no differentiation with the _fee function. For instance, TERM rewards are not aggregated at all while fee/yield rewards are aggregated even for staked positions. This will give a completely inaccurate return value. While it may seem at the first view irrelevant that TERM rewards for stakedPositions are completely disregarded since they don't fall into the core position of token0/1, they indeed should be considered part of the position in the case where token0/1 is TERM.
Recommendations	Consider refactoring the _fee function to properly differentiate between both position traits.
Comments / Resolution	Partially resolved by following the recommendation. It has to be noted that in the scenario where a in-range position has the isStaked trait, the TERM reward amount may not be accurately reflected up to block.timestamp. This is due to the fact that the TERM distribution is time-based instead of state-based.

Issue_22	Undripped yield is not incorporated into fees
Severity	**Medium**
Description	Yield is dripped linearly over the course of 24 hours as reward. This is currently not considered within the _fees function, which results in the yield between the last update and block.timestamp not being incorporated.
Recommendations	Consider implementing a mechanism which incorporates the due-yield. This requires an additional feature within the TerminalPool contract which returns the undistributed yield in a view-only manner. Additional validation time will be required if that is being implemented.
Comments / Resolution	Acknowledged.

Gauge

GaugeFactory

The GaugeFactory contract is responsible for deploying Gauges via the createGauge function which is triggered during the deployment process, outgoing from the Voter.createPool function.

Privileged Functions

- createGauge

No issues found.

Gauge

The Gauge contract is the central pivot point of the reward distribution mechanism. Each Gauge is related to a TerminalPool and has a corresponding FeeVault, YieldVault and BribeVault for voting casts and reward distributions

It handles several key components such as:

a) Accepting rewards from the Voter contract upon epoch flips via the notifyRewardAmount function

b) Setting and resetting shares during votes from users in all corresponding vault contracts

c) Handling the TERM distribution for stakedLiquidity during an epoch

d) Claiming revenue from the TerminalPool and distributing to the Vaults

Appendix: Rewards for stakedLiquidity

Upon each epoch flip, the Gauge receives the corresponding amount of TERM tokens based on the casted rewards in the previous epoch. These tokens will be linearly distributed based on the explained Appendix within the TerminalPool contract.

Core Invariants:

INV 1: The _sendRevenue function must first burn the token in case it is a RedeemableERC20 token

INV 2: The _sendRevenue function must allocate token0/1 to the FeeVault and YieldVault as voting rewards

INV 3: _notifyRewardAmount must always divide (rewardLeft + amount) by the remaining time until epoch ends

INV 4: rewardLeft / (epochEnd - blockTimestamp) must always be equal to rewardRate

INV 5: _getRewardGrowth must only be callable once per epoch, at the beginning.

INV 6: If _getRewardGrowth is called with stakedLiqudity = 0, calculated rewards must be aggregated in the rollover variable

INV 7: If _getRewardGrowth is called while block.timestamp > epochEnd, all leftover rewards must be distributed

INV 8: reward + rollover must always be calculated per unit of stakedLiquidity

INV 9: rollover must always be incorporated into the rewardGrowthDeltaX128 calculation

INV 10: _sendRevenue must always transfer underlyingToken to FeeVault/YieldVault

Privileged Functions

- setShares
- resetShares
- setUnstakedTax
- notifyRewardAmount
- notifyRewardBoost

Issue_23	Edge-case during epoch flip allows user to flash-theft a part of unallocated rewards
Severity	High
Description	Whenever an epoch is flipped, the notifyRewardAmount function is called which determines the new rewardRate. At the beginning of this function, _getRewardGrowth(0) is called which will increase the rollover variable by the reward amount which is due from [lastRewardUpdateTime; block.timestamp]. Note that this call will not only increase the rollover variable but will also set lastRewardUpdateTime = block.timestamp. This has the side-effect that any subsequent _getRewardGrowth call in the same block will return early: *// skip if second call in same block* *if (timeElapsed == 0) return 0;* Rewards are thus only distributed in the next block, if called with non-zero stakedLiquidity. This can be abused by a malicious user to bypass the rewardGrowthInside setting when liquidity is added and to flash-theft a part of the rewards which were accrued in the rollover variable. Illustrated: a) The vulnerable pool was updated last at TS = 10_000, the current TS = 20_000 and no swap/flash/etc has happened since back then. TERM rewards for this period have not yet been distributed. Timestamps here are only for illustrative purposes and do not necessarily match epoch periods whatsoever. b) The epoch flips and an attacker calls Voter.distribute with the corresponding gauge address which then calls Gauge.notifyRewardAmount and increases the rollover variable. In the same transaction, the user adds a large liquidity amount to the

	active range which means the user becomes the dominant liquidity provider in the active range. Due to the fact that TERM rewards are not updated because of the early return within _getRewardGrowth, the lastRewardGrowthInside value will be set to the current value without TERM rewards for the said timeframe c) In the next block, the attacker will simply withdraw liquidity again which now automatically updates rewards and thus the attacker will gain a majority of rewards from [lastRewardUpdateTime; block.timestamp].
Recommendations	Consider not returning early in case an update has already happened. Eventual side-effects must be considered.
Comments / Resolution	Resolved by following the recommendation.

Issue_24	Edge-case during epoch flip can result in loss of rewards for users by malicious actor
Severity	**Medium**
Description	Whenever an epoch is flipped, the notifyRewardAmount function is called which determines the new rewardRate. At the beginning of this function, _getRewardGrowth(0) is called which will increase the rollover variable by the reward amount which is due from [lastRewardUpdateTime; block.timestamp]. Note that this call will not only increase the rollover variable but will also set lastRewardUpdateTime = block.timestamp. This has the side-effect that any subsequent _getRewardGrowth call in the same block will return early. Rewards are thus only distributed in the next block, if called with non-zero stakedLiquidity. This can be abused by a malicious user to prevent a legitimate user from receiving rewards upon liquidity removal. Note how the root-cause is the same as within “Edge-case during epoch flip allows user to flash-theft a part of unallocated rewards” but a different exploit can be crafted out of it. Usually, if a user would withdraw (staked)Liquidity, TERM rewards between [lastRewardUpdateTime; block.timestamp] would be distributed to the user for the liquidity he owned during that time. This follows the standard pattern by updating the rewardGrowth and calculating rewardGrowthDelta, ... This can be exploited by an attacker via frontrunning the liquidity withdrawal and calling Voter.distribute for the corresponding reward (indeed, the epoch must be flippable here as well). That would then result in these rewards being casted in the rollover variable and the early return during the liquidity removal call would effectively not increase rewardGrowthGlobal which means the victim does not get these rewards when withdrawing liquidity, they

	are essentially forfeited and donated to all remaining liquidity providers.
Recommendations	Consider not returning early in case an update has already happened. Eventual side-effects must be considered.
Comments / Resolution	Resolved by following the recommendation.

Issue_25	Early return edge-case can be abused with strategic swaps
Severity	**Medium**
Description	The TerminalPool calls the Gauge function getRewardGrowth() to calculate new rewards to distribute to in-range staked liquidity positions. This function keeps track of the timeElapsed since it was last called, and always returns zero rewards if no time has passed: *if (timeElapsed == 0) return 0;* This logic is incorrect because it doesn't account for the rollover mechanism in the Gauge. The rollover variable tracks rewards that accumulate when there is zero staked liquidity, reserving those rewards for later distribution. An attacker can exploit this by ensuring there is no staked liquidity at the start of each block. If the first call to getRewardGrowth() in each block always observes zero staked liquidity, all emissions for that block are diverted into rollover for distribution in a later block. So, by always moving liquidity out of range using strategic swaps at the end of blocks, an attacker can delay reward distribution and influence its timing. However note that this strategy does incur swap fees and other MEV costs, which would likely outweigh any potential rewards, hence it is only marked as medium severity. It is important to distinguish this issue from the previous raised issues as this is a completely different attack-path. It is important that this issue was already raised by the previous auditor under “notes”. However, we are of the opinion that this issue exposes a medium severity.
Recommendations	Consider not returning early in case an update has already happened. Eventual side-effects must be considered.
Comments / Resolution	Resolved by following the recommendation.

Issue_26	Incorrect revenue distribution during _claimRevenue due to lack of yield update
Severity	**Medium**
Description	Whenever an epoch is flipped, the _claimRevenue function claims all gaugeRevenue from the TerminalPool contract via the collectGaugeRevenue function. The revenue consists of fees from swaps/flash and yield which is dripped. A problem because of the fact that the collectGaugeRevenue function does not invoke _updateYieldGrowth at the beginning. This means any yield which should be dripped between [lastDrip; block.timestamp] is not accrued and thus yield0/1 distribution to the Gauge/Vaults is lower than anticipated.
Recommendations	Consider calling _updateYieldGrowth at the beginning of the _collectGaugeRevenue function.
Comments / Resolution	Resolved by following the recommendation.

Issue_27	Users can influence corresponding underlying amount for epoch rewards
Severity	**Low**
Description	Whenever an epoch is flipped, the revenue is claimed from the TerminalPool and distributed to the Vaults. Since it is possible that the revenue is accumulated in RedeemableERC20 tokens, the nominal underlying amount is indeed then what is received by the burn call and distributed to the Vaults: *uint256 underlyingTotal = IRedeemableERC20(token).burn(address(this), total);* The current price of the RedeemableERC20 contract is what determines how much underlyingTotal will be received for the burned supply. If there is a deviation between the current price and the oracle price, users can influence whether more or less underlyingTotal is received. For example a user with a valid vote would avoid calling the update function while a user with no vote may call the update function in an effort to decrease the received underlyingTotal.
Recommendations	Consider implementing an incentive mechanism in the price oracle which incentivizes users to update the price more often.
Comments / Resolution	Acknowledged. The client intends to frequently update the price oracle.

Issue_28	Lack of incentive for liquidity providers of stakedLiquidity during epoch 0
Severity	**Low**
Description	Liquidity providers of stakedLiquidity do not receive any swap/flash/yield fee but receive TERM tokens based on the voting allocation. These tokens will be always distributed at the beginning of an epoch. Due the fact that there are no TERM tokens whenever epoch 0 is initiated, there is no incentive to add stakedLiquidity to the TerminalPool. If the TGE/epoch0 is happening nearly before thursday 00:00, the impact is low, otherwise in the worst case it can happen that no rewards for 7 days are received. Furthermore, there is the possibility for the team to provide TERM tokens via the notifyRewardBoost function which would solve this issue.
Recommendations	Consider acknowledging this issue and manually providing TERM rewards, if desired.
Comments / Resolution	Acknowledged. The clients intends to TGE right before the end of the epoch.

Issue_29	Rewards may be lost due to truncation in rewardRate calculation
Severity	**Informational**
Description	The rewardRate is calculated as follows: *rewardRate = rewardLeft / (epochEnd - timestamp);* This is prone to truncation, specifically if the reward token has 6 decimals. This issue is only rates as informational because the TERM token has 18 decimals.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Vaults

VaultsFactory

The VaultFactory contract is responsible for deploying the following contracts:

a) FeeVault
b) YieldVault
c) BribeVault

It furthermore sets the default allowed tokens which are:

a) FeeVault: token0/token1

b) YieldVault: token0/token1

c) BribeVault: token0/token1/TERM

Whereas if a token is the RedeemableERC20 token, the underlyingToken will be used.

Privileged Functions

- None

No issues found.

Vault

The Vault serves as a base contract for the following contracts:

a) BribeVault
b) FeeVault
c) YieldVault

It handles the voting process in form of share allocations and aggregates voting rewards.

Appendix: Reward Calculation

Usually, the _notifyRevenueAmount function is always called during an epoch flip, as it aggregates voting rewards to corresponding voters. More specifically, the pendingRevenue for each token is distributed among all shares:

> growthGlobalX128 = pendingRevenue * (2**128) / totalShares

Users can then claim their fair share as follows:

> userShares * growthGlobalX128 / (2**128)

Appendix: Undistributed Revenue

Whenever the _updateGrowthGlobal function is called for a new epoch while at the same time no shares are existent for this gauge, any provided amount will be allocated to the undistributedRevenue which can then be claimed by the Voter contract towards an arbitrary address.

Core Invariants:

INV 1: growthGlobalX128 must always incorporate the pendingRevenue per unit of shares

INV 2: collectRevenue must only be callable by authorized address

INV 3: collectRevenueForOwner must only be called by the Voter

INV 4: Vault.notifyRevenueAmount must increase pendingRevenue by the provided amount

INV 5: _updateGrowthGlobal must be called before any change in shares

INV 6: _updateGrowthFor must be called before any change in shares

INV 7: _updateGrowthGlobal must only be callable once per epoch

INV 8: Reward notification while totalShares = 0 must increase undistributedRevenue

INV 9: Whenever an epoch has surpassed, pendingRevenue must be distributed according to the totalShares amount before epoch flip

INV 10: _updateGrowthGlobal must always delete any consumed pendingRevenue

INV 11: Duplicate tokens within the allowedRevenue array are not allowed

INV 12: Revenue can only be collected to the token owner

INV 13: Revenue of a tokenId must only be increased based on growthDeltaX128

Privileged Functions

- collectRevenueForOwner
- collectUndistributedRevenue
- updateShares

Issue_30	Fundamental flaw in voting reward distribution will result in multiple issues
Severity	**Medium**

Description	In the standard VE system, rewards which are accrued during epoch 1 will be automatically distributed for votes which have been casted during epoch 1 during the end of an epoch. Some VE protocols still have a bug which allows claiming only in the subsequent epoch but this is irrelevant.
	
	The TLDR is that rewards/bribes which are accrued during epoch N must be corresponding to votes for epoch N.
	
	Within this particular scope the fee/yield rewards which are accrued during epoch 1 in the pool are notified during epoch 2 via the Gauge.notifyRewardAmount flow and the _claimRevenue/_sendRevenue flow (this is indeed only done once per epoch).
	
	These are then supplied to the FeeVault and YieldVault via the notifyRevenueAmount function. Due to the fact that the _notifyRevenueAmount function calls _updateGrowthGlobal before it increases pendingRevenue:
	
	function _notifyRevenueAmount(address token, uint256 amount) internal {
	if (!isAllowedRevenue[token]) revert NotAllowedRevenue(token);
	if (amount == 0) revert ZeroAmount();
	_updateGrowthGlobal();
	
	pendingRevenue[token] += amount;
	
	It will now actually set lastUpdateTime to a timestamp which is larger/equal as the epoch beginning. This means any subsequent call to _updateGrowthGlobal will result in an early return and pendingRevenue will only be distributed in the subsequent epoch.

	The rewards which were accrued during epoch 1 will thus be set aside for epoch 2, instead of being distributed in epoch 1. The TLDR of the technical explanation above is that rewards from epoch N (which should be distributed towards votes from epoch N) will be only distributed towards votes from epoch N+1. Among the fact that this is fundamentally wrong, there are three other important side effects to note: a) Votes from epoch 0 will never receive any rewards. b) Rewards which are accrued during epoch 1 and should be given to votes from epoch 1 will be forfeited and aggregated to undistributedRevenue if in epoch 2 no votes happened/all votes are reset (shares are zero) c) Inconsistency between Fee/Yield and Bribe rewards: This specific issue does not apply to any bribes which are provided via the BribesVault contract. Bribes which are provided during epoch 1 do actually invoke the notifyRevenueAmount function already during epoch 1 which then in turn increases the pendingRevenue. This means whenever the epoch flips to epoch 2, the pendingRevenue will be correctly distributed. This exposes an inconsistency between the mentioned reward types. Additionally, it must be mentioned that gauges which are created after the TGE has happened and after epoch 1, during the middle of an epoch, will already accrue fees during the said epoch. Moreover, it is then possible for anyone to call the Voter.distribute function for that corresponding gauge (since lastDistributed remains unset after creation. This will distribute any rewards which have been accrued since the creation of the TerminalPool until the most recent timestamp (the TS of the distribute call). These rewards will then be distributed for the votes towards this gauge once the epoch has flipped. Now the interesting part: rewards which have been accrued **after** the distribute function (but in the same epoch) up to the epoch flip **until the distribute function is called** during

	the new epoch, are then only distributed during the subsequent epoch. This means rewards during the epoch where the Gauge/TerminalPool/Vaults have been created will basically be distributed among two epochs in this specific edge-case. This also means that if a TerminalPool and its corresponding Gauge/Vaults are deployed at Thursday 00:01 AM while a user immediately calls Voter.distribute with this gauge address, no rewards are being transferred but lastUpdateTime will be set. This will also result in no rewards being distributed for voters during the first epoch, due to the already described root-cause. Rebase rewards are claimable for epoch 0.
Recommendations	This issue requires refactoring the whole voting approach. Therefore, we remain it open for the team to acknowledge the said side-effects (whereas the most impactful is the fact that votes from epoch 0 will not receive any fees/yield)
Comments / Resolution	Partially resolved by refactoring the distribution logic. Voters who cast votes for a gauge during epoch N will now have access to gauge revenue from epoch N once epoch N+1 starts. Here is the technical breakdown with an example: Example A: Epoch 0 to Epoch 1 > Users can provide liquidity and vote during epoch 0 > During epoch 0, fees and yield is accrued > Whenever epoch 0 is progressed to epoch 1, the Gauge will collect revenue from the Pool and distribute it to the Vault via notifyRevenueAmount > The notifyRevenueAmount function will increase pending revenue > After pending revenue is increases, the global growth is updated based on the increased revenue and the current VP > Therefore, votes during epoch 0 will receive fees from epoch 0 However, a specific issue was introduced which will result in a failure of full distribution during epoch 0. This is due to the fact that all fees which are accrued by the pool between deployment and first epoch flip will be marked as undistributed because the Vault is

	not synced with the Minter in terms of the current epoch. This means it is technically possible to update the epoch of the vault during epoch 0 which then follows the above described flow while at this point in time no votes are existing but rewards are. A trivial solution for this issue is to simply manually call flipEpochIfNecessary at the beginning of epoch 0. *Due to the refactoring, we recommend a follow-up audit of the epoch flip and reward distribution logic by another provider.*

Issue_31	Lack of full revenue distribution during epoch 0 due to unsynced state between Gauge and Minter
Severity	**Medium**
Description	During epoch 0 between the beginning of the epoch and the epoch end, the pool already accrues yield and fees. These are meant to be distributed to tokenIds which have voted during epoch 0. However, due to the fact that the Gauge is currently not synced to the Minter, the very first vote will already claim all fees and attempt to distribute these. At this point in time, there are no existing votes which means this distribution attempt will simply store these tokens in the undistributedRevenue mapping which means they will essentially be not distributed to voters.
Recommendations	Consider calling flipEpochIsNecessary manually at the week of the TGE.
Comments / Resolution	Acknowledged.

BribeVault

The BribeVault is an extension of the Vault contract and allows for accepting up to 16 tokens as bribes whereas the first three tokens are token0/1 and TERM. The notifyRewardAmount function is permissionless and thus allows anyone to provide allowed bribes.

Bribes which are provided during epoch N will be claimable for all voters during epoch N+1 based on their allocated VP during epoch N.

Core Invariants:

INV 1: Only the governor can add new revenue tokens

INV 2: Only valid bribe tokens can be notified

INV 3: Anyone can notify valid bribe tokens

INV 4: Can accept up to 16 different tokens

INV 5: Bribes distributed during epoch N must be claimablable once epoch N +1 starts by the allocated VP from epoch N.

Privileged Functions

- allowRevenue

Issue_32	Lack of support for transfer-tax tokens
Severity	**Informational**
Description	This contract is not compatible with transfer-tax tokens. If these token types are used for any purpose within the contract, this will result in down-stream issues and inherently break the accounting.
Recommendations	Consider not using such tokens.
Comments / Resolution	Acknowledged.

FeeVault

The FeeVault contract is an extension of the Vault contract and allows for accepting tokens as voting rewards, more specifically: token0/1, which are accrued as swap and flash fees.

The notifyRewardAmount function is only callable by the Gauge which happens once per epoch whenever the Voter calls the distribute function. It will then notify all fees which have been accrued during the past epoch.

Rewards which are accrued during epoch N will be claimable for all voters during epoch N+2 based on their allocated VP during epoch N +1.

Core Invariants:
INV 1: Only the gauge can call the notifyRewardAmount function.

Privileged Functions

- notifyRewardAmount

Issue_33	Redundant usage of FeeVault and YieldVault
Severity	**Low**
Description	Currently, the FeeVault and the YieldVault contracts accept exactly the same tokens. The only difference is the fact that a part of the yield within the YieldVault will be distributed towards the YieldRouter. This means, it would be possible to simplify the whole process and just use one Vault instead of two.
Recommendations	Since this has already been fully audited by another provider, we do not recommend such intrusive changes anymore, unless they are absolutely necessary.
Comments / Resolution	Acknowledged.

YieldVault

The YieldVault contract is an extension of the Vault contract and allows for accepting tokens as voting rewards, more specifically: token0/1, which are accrued as yield rewards.

The notifyRewardAmount function is only callable by the Gauge which happens once per epoch whenever the Voter calls the distribute function. It will then notify all fees which have been accrued during the past epoch.
A part of the yield is then transferred to the YieldRouter which gives the governor the freedom to distribute them as bribes. The exact amount which is transferred to the YieldRouter is based on the protocolShare variable.

Rewards which are accrued during epoch N will be claimable for all voters during epoch N+2 based on their allocated VP during epoch N +1.

Core Invariants:

INV 1: amount * protocolShare / MAX_BPS must be transferred to the YieldRouter

INV 2: Only the yieldGovernor can call the setProtocolShare function

INV 3: Only the Gauge can call the notifyRevenueAmount function

INV 4: Must only receive fees in form of underlying tokens

Privileged Functions

- setProtocolShare
- notifyRevenueAmount

No issues found.

YieldRouter

The YieldRouter is a standalone contract which is not tied to any specific Gauge/TerminalPool. It receives various different tokens by the YieldVault for each corresponding TerminalPool which can then be distributed via the bribe function by the yieldGovernor towards desired BribeVaults

Core Invariants:

INV 1: Only the yieldGovernor can call the bribe function

Privileged Functions

- initialize
- setYieldGovernor

Issue_34	Governance Issue: Funds can be transferred out
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. The yieldGovernor can bribe and vote to an empty gauge in the same transaction just in the last block before epoch is flipped which would then result in the yieldGovernor receiving the full bribe amount.
Recommendations	Consider ensuring that the yieldGovernor is sufficiently trusted.
Comments / Resolution	Acknowledged.

Redeemble

RedeemableChecker

The RedeemableChecker contract is a simple helper library which can be used to query whether a target contract exposes the corresponding interfaces to a RedeemableERC20 contract.

Core Invariants:

INV 1: A contract is only considered as RedeemableERC20 if the ERC165 and REDEEMABLE_ERC20 interfaces are supported AND the ID_INVALID interface is not supported.

Privileged Functions

- none

No issues found.

RedeembleERC20

The RedeemableERC20 contract is a wrapper contract for tokens which increase natively in price such as sUSDe. This price increase is reflected by a rebasing of the supply which basically converts a rebasing token with an ever increasing price to a rebasing token with an ever increasing supply.

This functionality is then used to distribute rebase rewards in the TerminalPool contract.

Appendix Mint & Burn

Users can mint RedeemableERC20 tokens by providing the underlying token using the following formula:

> amount * 1e8 / price

Similarly, users can burn the RedeemableERC20 token in an effort to receive the underlying token back:

> burnedAmount * 1e8 / price

Appendix: Yield Update

Whenever the price is increased, this means that users will get less of the underlying token if the supply is burned. In an effort to ensure that the underlying amount is matched with the totalSupply, the _updateYield function calculates how much yield would be missing if the totalSupply stays the same but the price is updated:

> oldTotalPrincipal = totalSupply * 1e8 / oldPrice
> newTotalPrincipal = totalSupply * 1e8 / newPrice
> yield = oldTotalPrincipal - newTotalPrincipal

Afterwards, this yield is allocated as reward for users to claim and once a claim happens, it is multiplied with the current price which then exactly matches the corresponding yield for a user:

> yield * price

This principle ensures that all owners will always receive the corresponding supply amount to match the underlying provided amount. Even if a user claims after another price update, the yield * price multiplication ensures that the user will always receive the correct token supply to cover the underlying amount.

Core Invariants:

INV 1: Transfer is not allowed if tx.origin has updated the price in the same block

INV 2: Within _updateYield, oldTotalPrincipal must be >= newTotalPrincipal

INV 3: Mint must round up required underlying amount

INV 4: Burn must round down received underlying amount

INV 5: Starting price must always be 1e8

INV 6: After _updateYield is called and all tokens are minted, totalSupply * 1e8 / price must be equal to principal amount (+- rounding)

INV 7: _updateFor must be called before balanceOf increases

INV 8: Burn/mint must call _beforeTokenTransfer

INV 9: (totalSupply / price) + aggregated redeemable.yield + unupdated yield must always be corresponding to underlyingAmount

Privileged Functions

- setPrice

Issue_35	Initial price of 1e8 can result in a loss for liquidity providers
Severity	**Medium**
Description	As we have explained, the main goal for the introduction of this contract is to share the yield among liquidity providers instead of keeping it for the user. We have already elaborated an issue where a user might lose some tokens if liquidity is deposited while the price is unupdated, as now the price update will result in the yield being distributed among the liquidity providers instead towards the user. This issue is inflated due to the fact that the RedeemableERC20 token is always deployed with a price of 1e8. If now however the underlying token is > 1e8, which is most likely always the case, as sUSDe is right now worth 1.16e8, there is a large price discrepancy after deployment which is only offset during the initialize call. This can happen if the PythOracleProxy contract as well as the RedeemableERC20 contract is deployed and whenever liquidity is added before PythOracleProxy.initialize is called. Moreover, the PythOracleProxy contract must be deployed before the RedeemableERC20 contract since the RedeemableERC20 constructor requires a valid oracleProxy address. This outrules the PythOracleProxy deployment and initialization in the same transaction. Thus, this scenario is technically possible since there is no explicit safeguard which requires the PythOracleProxy contract to be initialized before any pool interactions can be made.
Recommendations	Consider preventing mints in the RedeemableERC20 contract if the corresponding PythOracleProxy contract is not initialized. This can be done by a simple check if the RedeemableERC20 variable within the PythOracleProxy is still address(0).
Comments / Resolution	Resolved by specifying the initial price in the RedeemableERC20 constructor.

Issue_36	Potentially skipped yield during price update
Severity	**Low**
Description	The _updateYield function incorporates the following condition: *if (* *newTotalPrincipal * newPrice < totalSupply() * MAX_BPS &&* *newTotalPrincipal + 1 <= oldTotalPrincipal* *) {* *newTotalPrincipal++;* *}* This was developed to prevent undercollateralization. An important observation here is that after the increase of newTotalPrincipal, it may happen that newTotalPrincipal = oldTotalPrincipal. In such a scenario, yield becomes zero while the price was increased. Indeed, this seems more as a hypothetical issue because we were not able to reproduce that exact edge-case. However, in the scenario where someone is able to reproduce it, one could forcefully exploit small price updates to result in no tokens being minted.
Recommendations	Consider keeping this potential condition in mind.
Comments / Resolution	Acknowledged.

Issue_37	Griefing vector within burn function
Severity	**Informational**
Description	The burn function accepts the amount parameter which is corresponding to the RedeemableERC20 amount burned by the user. The user will then receive underlying amount based on the current price: > amount * 1e8 / price Thus, if a user wants to burn 100e18 tokens with the current price being 1e8, he receives 100e18 underlying tokens. This can be griefed by another user calling the update function on the PythOracle before, resulting in a price increase to 1.1e8. The user would then effectively only receive 90.90e18 tokens in return and is required to redeem and burn the leftover in another transaction.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Oracle

PythOracleProxy

The PythOracleProxy is a casual oracle handler contract which fetches the price from a PythOracle via the getPriceNoOlderThan function, then takes the lower bound based on the confidence interval and updates the price. It furthermore exposes freeze and unfreeze functionality which can be used by governance in emergency situations.

Appendix: Price Conversion

The price is fetched from the Pyth oracle via the getPriceNoOlderThan function. This function returns the price with its corresponding exponent. If for example the return value is 1.1e8 and the exponent is -8, this means the price will be 1.1.

Appendix: Maximum Increase Calculation

The contract has a built in safeguard which is against short and long term price fluctuations. This is achieved via the maximumWeeklyRate variable whenever the price is updated.

The maximum allowed increase is calculated based on the lastPrice, the time elapsed since the last update and the maximumWeeklyRate downscaled to seconds:

> (lastPrice * weeklyRate * timeElapsed) / (100000000 * 1 week)

If the (positive) difference between the new price and the old price is larger than the maximum allowed increase, the price increase is simply clamped to the maximum increase.

Core Invariants:

INV 1: The weekly price increase must not be larger than maximumWeeklyRate

INV 2: Price must be reflected with 8 decimals

INV 3: Price must never decrease

Privileged Functions

- initialize
- setEmergencyAdmin
- freeze
- unfreeze

Issue_38	Bypass of maximumWeeklyRate due to compound effect in rate calculation
Severity	**Medium**
Description	It is expected that the token price cannot increase more than the maximumWeeklyRate each week. If for example the rate is 0.1e18, it is expected that the token cannot increase more than 10% during each week. Due to the compounding nature of the price calculation, this is actually bypassable via regular update calls: Illustrated: **Scenario 1:** price = 1e8; rate = 0.1e8 (10%), timeElapse = 1 week > maximumIncrease = 1e8 * 0.1e8 / 1e8 > maximumIncrease = 0.1e8 > newPrice = 1.1e8 **Scenario 2:** first: price = 1e8; rate = 0.1e8, timeElapsed = 3.5d > maximumIncrease = 1e8 * 0.1e8 * (86400*3.5) / (1e8 * (86400*7)) > maximumIncrease = 0.05e8 > newPrice = 1.05e8 second: price = 1.05e8; rate = 0.1e8; timeElapsed = 3.5d > maximumIncrease = 1.05e8 * 0.1e8 * (86400*3.5) / (1e8 * (86400*7)) > maximumIncrease = 0.0525e8 > newPrice = 1.10525e8

	We effectively bypassed the 10% and achieved a 10.525% increase. The more often a compound happens, the higher the potential increase.
Recommendations	Consider either refactoring the whole formula or accounting for the compounding effect when determining the maximumWeeklyIncrease value (choosing a slightly lower value to counter this scenario).
Comments / Resolution	Acknowledged.

Issue_39	maximumWeeklyRate can backfire
Severity	**Low**
Description	The implementation of the maximumWeeklyRate is an important circuit breaker mechanism in case there are unnatural changes in the price. However, it may result in a fundamental lag behind the real price.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_40	Important variables are immutable
Severity	Low
Description	Variables such as the maximumWeeklyRate are immutable and can never be changed. If for any scenario the dynamics of the underlying token change, it will be impossible to adjust for that due to the fact that the weekly rate can never be changed. Moreover, the priceFeedId cannot be changed which can result in issues if the feed is permanently not available. This issue is even more amplified due to the fact that the RedeeambleERC20 contract permanently points to this pythOracle address
Recommendations	Consider implementing functionality which allows for changing important variables.
Comments / Resolution	Acknowledged.

Issue_41	Lack of fallback mechanism in case pyth price is unretrievable
Severity	Low
Description	Currently, the price is fetched via the getPriceNoOlderThan function from the Pyth oracle. If in any specific scenario the Pyth oracle is not able to return the current price, this effectively means that no price update is possible. This will result in a temporary (or even permanent) prevention of yield distribution.
Recommendations	Consider implementing a fallback mechanism. Moreover, it might make sense to include a sequencer uptime check.
Comments / Resolution	Acknowledged.

Issue_42	Price initialization is done using admin provided parameter
Severity	**Low**
Description	During the initialize function, lastPrice is set to the provided initialPrice parameter. This is under most circumstances not a problem. However, best-practice would be to just use the current pyth price.
Recommendations	Consider if it is desired to change this mechanism, otherwise this issue can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_43	Confidence granularity may be insufficient
Severity	**Informational**
Description	The contract has the following confidenceThreshold check: *if (* *currentPrice.conf != 0 &&* *currentPrice.price / int64(currentPrice.conf) < confidenceThreshold* *) revert InsufficientConfidence();* Since price is not scaled by any multiplier, the confidenceThreshold must be any nominal value between 1 and 100. This may result in a slight flexibility limitation in case the confidenceThreshold should be a floating point number
Recommendations	Consider if it is desired to scale the price in an effort to increase the granularity. Otherwise this issue can be acknowledged.
Comments / Resolution	Acknowledged.

TerminalFi - Scope 2

Project	TerminalFi - Scope 2
Website	terminal.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/terminal-finance/contracts/tree/756cce7f34ee81592af955968c8d982fc3089171
Resolution 1	https://github.com/terminal-finance/contracts/blob/3ef5a12896cecb2f3298a7f1fd889a5e566a2db4/

Voting

Voter

The Voter contract is the core contract for the deployment process, reward distribution and voting procedure. It essentially allows users to vote using their owned or approved tokenId for one or multiple desired pools during epoch N. Once epoch N is then flipped, TERM rewards are distributed by the Minter contract to this contract and allocated towards the different pools based on their casted voting allocation during epoch N. Voting can be done via the vote function once per epoch while the poke function is permissionless and callable multiple times per epoch. It is furthermore also possible to reset votes once per epoch via the reset function.

Votes during epoch N do not need to be repeated during epoch N+1 as a vote and its corresponding VP remains valid for all subsequent epochs.

Appendix: Deployment Flow

Contrary to the standard VE protocol, pool deployment is not permissionless. Instead, only pools for whitelisted assets can be created by the governor. Multiple different components are related to one single TerminalPool, such as:

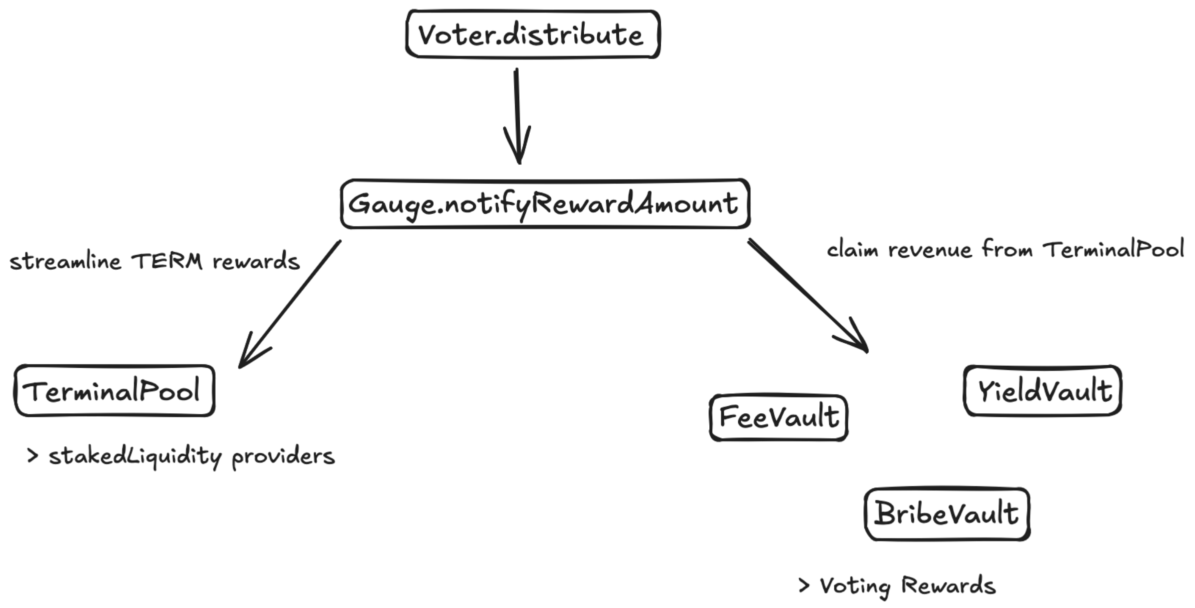
a) Gauge
b) FeeVault
c) YieldVault
d) BribeVault

These contracts are deployed during the createPool function, right after the TerminalPool has been deployed and remain always tied to the same TerminalPool. They are responsible for TERM rewards and voting rewards. A more in-depth explanation can be found in the corresponding sections, above in the report.

Appendix: Reward Distribution

TERM rewards are accrued each epoch flip via the Minter contract and will be reflected in the index increase. This behavior is well-described in the below appendix.

The actual distribution process only happens via the permissionless distribute function which is only callable once per epoch and distributes allocated rewards to the Gauge. These rewards are then streamlined linearly over the course of the epoch to stakedLiquidity providers.



Appendix: Reward Notification

Whenever an epoch is flipped, the Minter contract invokes the notifyRewardAmount function which transfers the weekly TERM emissions in and increases the index variable, which represents an accumulating reward ratio, scaled by 1e18.

Whenever notifyRewardAmount transfers new tokens in, the contract increments index by:

> amount * 1e18 / totalWeight

Later, each gauge's _updateFor(gauge) compares the current index to its own supplyIndex[gauge]. The difference, multiplied by that gauge's share of total weight, is credited to claimable[gauge]:

> delta = index - supplyIndex[gauge]
> share = (weights[pool] * delta) / 1e18

Then supplyIndex[gauge] is set to the new index. This ensures rewards are allocated proportionally to each gauge's stored weight over time.

Core Invariants:

INV 1: Within _vote duplicate pools must not be allowed

INV 2: When a tokenId is used for a vote, it must be marked as voting = true

INV 3: When a tokenId is reset, it must be marked as voting = false

INV 4: The vote and reset functions can only be called once per epoch

INV 5: A pool can only be created for whitelisted tokens

INV 6: Whenever notifyRewardAmount is called, index must be increased by amount * 1e18 / totalWeight

INV 7: Anyone can call the poke function

INV 8: _updateFor must always be triggered before any vote allocation change

Privileged Functions

- notifyRewardAmount
- initialize
- setGovernor
- whitelist
- collectUndistributedRevenue
- setEmergencyCouncil
- killGauge
- reviveGauge
- createPool
- collectUndistributedRevenue

Issue_44	Sophisticated edge-case allows for keeping initial VP indefinitely
Severity	**High**
Description	The contract is developed in such a manner that once a vote is casted it will always remain casted unless the user votes again or resets. This means the initial VP at the time of the vote will always get used for VP allocation purposes, including share setting in all Vault contracts. In itself, this is a medium severity due to the fact that the poke function is permissionless and in the worst case a script can be set up which identifies such unupdated positions and pokes these. Even though this is not an ideal scenario, it helps to mitigate this issue. However, during our in-depth assessment we found a sophisticated way to brick the poke function from being called. This means a malicious user can execute a vote while ensuring that the VP will never decrease because nobody can poke this vote. The root-cause for this exploit lies within the _vote function and the fact that it is possible to execute votes with zero amounts for pools due to this arithmetic operation: *uint256 _poolWeight = (_weights[i] * _weight) / _totalVoteWeight;* This can be trivially done by providing a super small element in the weights array with any pool while at the same time providing regular weights which are way larger which results in _totalVoteWeight becoming larger than _weights[i] * _weight and thus a _poolWeight of zero, to execute a normal vote. In that scenario, it will not revert but simply create a zero-weight vote for a pool which is then mirrored in the votes mapping: *votes[_tokenId][_pool] += _poolWeight;* The creation of such a zero-weight vote for a pool then prevents the poke function from being called, as it will cast all weights done by

	this tokenId in the _weights[] array:
	
	for (uint256 i = 0; i < _poolCnt; i++) {
	_weights[i] = votes[_tokenId][_poolVote[i]];
	}
	
	which simply mimics previous votes and invokes the _vote function.
	
	A sanity check at the beginning of the _vote function is then executed which results in a revert:
	
	for (uint256 i = 0; i < _poolCnt; i++) {
	if (_weights[i] == 0) {
	revert ZeroWeightVote();
	}
	
	Following this exploit, a user will be able to vote with manipulated VP, even if a tokenId is technically already unlocked and has zero VP.

Recommendations	Consider ensuring that _poolWeight cannot be zero:
	
	// if (_poolWeight == 0) {
	// revert ZeroWeightVote();
	// }

Comments / Resolution	Resolved by following the recommendation.

Issue_45	Incorrect killGauge implementation will result in multiple side-effects
Severity	High

Description	The killGauge function allows emergencyCouncil to kill a gauge which essentially set isAlive = false and thus does not allow for this gauge to receive rewards.
	
	Due to an insufficient implementation of the said function, there are multiple side-effects.
	
	a) Killing will immediately result in undistributed funds being locked as the claimable[gauge] mapping is simply erased without any refunding.
	
	b) If the index is increased while claimable[gauge] has not been updated, claimable[gauge] will be updated as soon as the _updateFor function is called. If now a gauge is killed and _updateFor is called, the index increase will not be used, due to the if condition never being triggered:
	
	`if (isAlive[_gauge]) {`
	`    claimable[_gauge] += _share;`
	`}`
	
	rewards are thus permanently locked.
	
	c) Voting increases the totalWeight variable which is used as divisor during the notifyRewardAmount function:
	
	`uint256 _ratio = totalWeight > 0`
	`? (amount * 1e18) / totalWeight`
	
	Whenever a gauge is killed, this gauge won't receive any further reward allocation. However, totalWeight is not decreased and thus a fraction of rewards are always wasted/locked unless all votes for this corresponding pool are reset.

	d) Voting for killed gauges is always possible which will further inflate c)
Recommendations	Consider refactoring the usage of the killGauge function. Since it is a complex interplay between different functions and variables, such a change must be carefully validated with additional time. Optionally, this issue can also be fixed by introducing a recoverERC20 function which allows the withdrawal of any locked TERM tokens. While this fix is not ideal, it at least does not introduce potential side-effects.
Comments / Resolution	Resolved. The logic for this has been refactored. We will comment on the specific mentioned scenarios: a) Killing will now distribute claimable funds to the Minter b) A new condition within _updateFor has been introduced which will transfer out funds to the Minter in the specifically mentioned scenario. c) Same fix as b) d) Same fix as b) Specifically side-effects may occur if a gauge is revived again (in the same epoch). For example if a gauge is not updated it will accumulate a delta in supplyIndex which would then usually be transferred to the Minter once updated. However, if revived after a couple of epochs without updating in between, voters will receive all these rewards. *We recommend a follow-up audit of this mechanism. In the meantime the protocol can be deployed but this feature should be used very carefully. Full confidence is not reached during the mitigation review.*

Issue_46	Poke function can be used to decrease VP from other users while attacker can profit from it
Severity	**Medium**
Description	As already explained, the poke function serves as permissionless function to adjust other user's VP in case their votes are stale for a certain time and still allocate the initial VP. It has to be noted that the VP is not only decreasing each epoch but linearly with each passed second which means that this function can forcefully decrease VP of other users by simply calling it during the last block of an epoch. Following that strategy a user can vote at the beginning of an epoch for a specific gauge and then poking **all other** votes right before the end of the epoch for the same gauge to decrease their VP in an effort to gain more rewards.
Recommendations	A fix for this issue would mean making the poke function permissionless and this also means the whole vote flow must be refactored as the poke function can now not be used anymore to correct votes. An idea would indeed be to allow only the tokenId owner/approved address to poke the tokenId while at the same time allowing a BOT role to call it for correction purposes.
Comments / Resolution	Acknowledged.

Issue_47	Permanently locked rewards if no votes during an epoch happened
Severity	**Medium**

Description

The notifyRewardAmount function calculates the index increase as follows:

```
uint256 _ratio = totalWeight > 0
? (amount * 1e18) / totalWeight
: (amount * 1e18);

if (_ratio > 0) {
index += _ratio;
}
```

In the scenario where there are no votes, it will simply increase the index by amount * 1e18. However, due to the way how gauges are being updated, it will simply always update the index to the new one if there are no existing votes:

```
if (_supplied > 0) {
uint256 _supplyIndex = supplyIndex[_gauge];
uint256 _index = index; // get global index0 for accumulated distro
supplyIndex[_gauge] = _index; // update _gauge current position to global position
uint256 _delta = _index - _supplyIndex; // see if there is any difference that need to be accrued
if (_delta > 0) {
uint256 _share = (_supplied * _delta) / 1e18; // add accrued difference for each supplied token
if (isAlive[_gauge]) {
    claimable[_gauge] += _share;
}
}
} else {
supplyIndex[_gauge] = index; // new users are set to the default global state
```

	``` }``` All rewards which are distributed in such a scenario are permanently locked. This issue has only been rated as medium instead of high severity due to the fact that such a situation is very unlikely (but still technically possible).
Recommendations	Consider implementing an if-condition and only transfer funds in if totalWeight > 0. One can consider the SwapX Voter contract which handles this scenario properly.
Comments / Resolution	Resolved by following the recommendation.

Issue_48	onlyNewEpoch modifier is time-based instead of epoch based
Severity	Low
Description	The onlyNewEpoch modifier enforces a week-based check by comparing block.timestamp / DURATION (where DURATION = 1 week). This means that if we're still within the same 7-day window (e.g., epoch 1) and the time crosses a boundary like Thursday 00:00, any subsequent vote in still counts for epoch 1 until the epoch is flipped, while it actually was already done in the subsequent epoch (time-based). If the epoch is now flipped, the user will have no chance to vote for the new epoch and will be trapped with the previous vote. This has only been rated as low instead of medium since the epoch is casted anyways for epoch 2 as well. The only downside for the user is that he will not be able to vote for different pools during epoch 2 than he did for epoch 1.
Recommendations	Consider acknowledging this issue as fixing it would require rewriting the modifier logic.
Comments / Resolution	Acknowledged.

Issue_49	Missing distributeAll function
Severity	**Low**
Description	Currently, there is no existing distributeAll function which loops over all gauges. In the scenario where one gauge is not distributed in one epoch, stakedLiquidity providers will not receive any rewards for the epoch and voting rewards will also not be allocated for the said epoch.
Recommendations	Consider implementing a distributeAll function which loops over all gauges (additionally to the standard distribute function).
Comments / Resolution	Acknowledged.

Issue_50	Zero amount revert will break the contract after a long time
Severity	**Informational**
Description	The notifyRewardAmount function explicitly reverts if the amount is zero. Since the weeklyAmount is gradually decreasing, there is a very distant point in the future where this condition will be triggered and thus updateEpoch permanently reverts, which results in gauge fees being locked in the pool because the epoch can never be updated and thus distribute will never call Gauge.notifyRewardAmount. This issue is only rated as informational since this will so far far in the future that it can be disregarded.
Recommendations	Consider removing the zero revert.
Comments / Resolution	Resolved by following the recommendation.

VotingEscrow

The VotingEscrow is one of the most fundamental contracts in a VE scope. It allows users to execute a different range of operations, starting from locking their TERM tokens for a certain duration in exchange for a VE tokenId which represents voting power, over splitting and merging tokenIds. Users can lock their TERM tokens for a maximum time of 2 years which will yield the maximum VP which can be used to vote for pools within the Voter contract as well as to propose and influence governance decisions via the L2Governor contract.

The contract uses a curve inspired VP calculation algorithm which will be described in the appendix below. The contract furthermore incorporates the HistoricalSupply and HistoricalTokenSupply libraries which are used for fetching the overall VP and and individual VP of a tokenId. These two libraries are however **not included** in the audit scope.

Appendix: Voting Power Logic

The _checkpoint function exposes the fundamental algorithm which ensures that the voting power logic is always accurately updated. It is forked from the widely used _checkpoing logic in most VE architectures with a small adjustment which ensures that only one checkpoint per timestamp is existing.

Initially the VP can be as high as the nominal locked token amount, if locked for 2 years. If however tokens are not fully locked for four years, the initial VP will be calculated as follows:

*amountToLock * (lockEndTs - currentTs) / 2 years*

This means if a user only locks his tokens for 1 year, the VP will initially be 50% of the locked amount and decay linearly with the increase of currentTs.

To facilitate this mechanism, a sophisticated algorithm was implemented which keeps track of:

A tokenId's point at specific epochs (usually whenever the tokenId was deposited or manipulated). If for example a tokenId is created via a lock, the following variables are saved:

tokenPointHistory[epoch]

- bias: slope * (lockedEndTs - currentTs)
- slope: amountToLock / 2 years

- ts: timestamp of tokenId creation
- blk: block.number of the tokenId creation

Whenever now the VP of this tokenId is fetched, this is simply done as:

*lastPoint.bias -= slope * (currentTs - lastPoint.ts)*

This approach reflects the decaying VP

The totalSupply of all tokenIds. This is facilitated in:

pointHistory[epoch]

- bias: decreased over time in similar fashion as above, aggregates bias from all tokenIds
- slope: aggregates slopes from all tokenIds, decreased whenever a tokenId's lock has surpassed
- ts: timestamp of last update for global supply
- blk: block.number of last update for global supply

The usual epoch duration is one week and this algorithm ensures that a tokenId always displays the correct VP, the total aggregated VP forms the totalSupply and the totalSupply and tokenId VPs are steadily decreasing.

Below we will re-iterate all state variables and explain them:

pointHistory[epoch] = Point

This mapping assigns the Point struct to a corresponding epoch. It simply stores all relevant information for each epoch:

> bias: total VP
> slope: total slope
> ts: timestamp for this epoch
> block: block number for this epoch

This value is influenced whenever tokenIds are modified and simple during the global update which decays the VP and removes slope from now unlocked tokenIds.

Each _checkpoint call (once per block) will then create a new epoch and save the global checkpoint into this mapping. Upon total supply consultation, a binary search is executed to find the nearest checkpoint before or equal the lookup timestamp and then extrapolate similar to the _checkpoint logic itself the correct total supply

tokenEpochs[tokenId] = index

The tokenEpochs mapping provides information how often a tokenId has already been modified. If for example a tokenId has just been created, the corresponding index will be 1. If it now has been changed, for example the amount increases or the unlockTime is being extended, the index will be 2, etc.

tokenPointHistory[tokenId][epoch] = Point

The tokenPointHistory mapping provides information about the specific checkpoint of a token at a corresponding epoch:

> bias: total VP
> slope: total slope
> ts: timestamp for this epoch
> block: block number for this epoch

If for example a tokenId has just been created, the Point will include the current VP as well the slope. If then the tokenId is modified, a new checkpoint for this tokenId with the corresponding epoch is written. Using this checkpoint logic it will then be possible to fetch the accurate VP for a user based on a provided timestamp. This will done via a binary search implementation which fetches the nearest epoch before or equal the desired timestamp and then extrapolates the VP based on the slope and the time difference between the checkpoint timestamp and the search timestamp.

slopeChanges[timestamp]

The slopeChanges mapping aggregates information which is corresponding to slopes from tokenIds and their corresponding unlock time. For example if a tokenId is created with an amount of 100e18, the corresponding slope for this tokenId will be 1.58e12 (100e18 / 2 years). This slope is then reversed in the sign and added to the slopeChanges[endTime] mapping. This mapping is then used during the update of the global checkpoint. Whenever a week has been passed where tokenIds are unlocked, this will thus decrease the slope for this global checkpoint, reflecting that these tokenIds are no longer contributing to the checkpoint and thus reducing the decay speed due to the decrease of slope.

Appendix: Modification of tokenId

Whenever a tokenId has been minted via the createLock/createLockFor function, the following modifications can be done:

a) Lock duration can be extended
b) Amount can be increased
c) tokenId can be split in two new tokenIds
d) tokenId can be merged into another existing tokenId

Core Invariants:

INV 1: A tokenId can only been withdrawn if block.timestamp >= lock.endTime

INV 2: Withdraw, merge and split is not allowed if a tokenId is actively used in voting

INV 3: During merge, the larger of both endTimes must be used

INV 4: LockedBalance.slope must always be set to amount / 2 years

INV 5: LockedBalance.bias must always be slope * timeUntilEnd

INV 6: Modification of tokenId must always adjust VP of current global checkpoint

INV 7: slopeChanges must always point to thursday 00:00 for the corresponding epoch end

INV 8: epoch is increased every time _checkpoint is called

INV 9: global slope is only decreased if a week has passed

INV 10: During vote, the pool address is provided which is then used to derive the gauge address

Privileged Functions

- setTeam
- setArtProxy
- allowSplit
- initialize

Issue_51	No expiration check in merge() function
Severity	**Low**
Description	The merge() function allows merging even if one or both tokens are expired. Most other functions block expired tokens, so this behavior is inconsistent. While this doesn't seem to create an exploit (since the _checkpoint() function correctly removes both tokens from the global slope/bias if they are expired), it does seem to contradict some comments in the code, for example: *// newLocked.end > block.timestamp (always)* *_checkpoint(tokenId, oldLocked, newLocked);* Most of the time, exploits happen due to arbitrary user inputs or users invoking functions which are not meant to be invoked by users, one can argue that a large user flexibility is a great seed for exploits. Therefore, at BailSec, we are of the opinion that codebases should never provide more user flexibility than necessary during the normal business logic.
Recommendations	Consider adding a check in merge() to prevent merging tokens that are expired.
Comments / Resolution	Acknowledged.

Issue_52	Unreset tokenId split approval in case of transfer.
Severity	**Low**
Description	Whenever a tokenId is being transferred, this will never reset the split allowance and can thus result in a split even if it is owned by a different address.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_53	Transfer of tokenId without previous claiming will result in forfeiting rewards to new owner
Severity	**Informational**
Description	Currently, whenever a tokenId is transferred, voting and rebase rewards are not automatically claimed. This will result in the current owner essentially forfeiting rewards to the new owner. In the scenario where the NFT is listed on a marketplace, this could also result in phishing buyers by frontrunning a purchase and claiming rewards, such that buyers which intended to buy a tokenId with tied rewards will only receive the tokenId without rewards. This is generally considered as a design-choice but it is still important to point it out, especially to users.
Recommendations	Consider making it clear that rewards are tied to the tokenId instead of the owner.
Comments / Resolution	Resolved by adding a comment in the code detailing this behavior.

Issue_54	Potential SafeCast reverts
Severity	**Informational**
Description	The contract uses safeCast on multiple occasions. Just to note one example: *LockedBalance memory newLocked = LockedBalance({* *amount: oldLocked.amount + SafeCast.uint256ToInt128(amount),* *end: unlockTime == 0 ? oldLocked.end : unlockTime* *});* In the scenario of a high token supply, this could revert in the downcasting. Specifically for the number 170141183460469231731e18. This issue is only informational since the TERM token will never reach such a high supply.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Token

MerkleClaim

The MerkleClaim contract is a simple distribution contract which allows entitled addresses to claim liquid and locked TERM tokens via the claimLiquid and claimLock function.

For validation purposes, a merkle tree is used which is provided upon deployment and immutable. The contract will mint TERM tokens directly via privileged call to the Term contract either directly to the recipient or will lock it for the recipient via VotingEscrow.createLockFor.

Core Invariants:

INV 1: Within claimLiquid, the leaf must exist out of [to, amount]

INV 2: Within claimLocked, the leaf must exist out of [to, amount, lockDuration]

INV 3: Within claimLiquid and claimLocked, the to address can receive funds only once

Privileged Functions

- none

Issue_55	Unclaimed rewards for tokenIds which exist longer than one epoch
Severity	**Low**
Description	Whenever a claim happens, the current tokenId is split in an effort to create a new tokenId with the corresponding balance for the claim. If now a tokenId has “survived” an epoch flip, it is entitled for rewards within the RewardsDistributor contract. This entitlement will be lost if the tokenId is then again split.
Recommendations	Consider implementing a governance function which allows for claiming rewards from the RewardsDistributor. *Furthermore, we recommend a follow-up audit due to the refactoring of the codebase.*
Comments / Resolution	Acknowledged.

Issue_56	Off-by-one error between onlyBeforeDeadline and onlyAfterDeadline
Severity	**Informational**
Description	Currently, there is a time when both modifiers are true: *block.timestamp > claimDeadline* This means at this time it is possible to claim and also to recover unclaimed funds. This can result in operational side-effects.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_57	Incompatibility with IERC721Receiver
Severity	**Informational**
Description	Currently, the contract does not expose the required interface for receiving ERC721 tokens via the “safe” functions. Therefore, specific transfers will revert.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_58	The “to” address can only receive claim once
Severity	**Informational**
Description	Currently, the hasClaimedLiquid and hasClaimedLocked maps the “to” address to true whenever a claim has happened. This will prevent a different claim with another leaf for the “to” address to revert.
Recommendations	Consider if there is the possibility of allowing an address to receive different claims, for example a different amount with a different lockDuration. If that is a desired feature, then the validation must be adjusted to mark the leaf as claimed. However, most likely this issue can be acknowledged safely.
Comments / Resolution	Resolved with a refactor of MerkleClaim, which now supports multiple claims for the same “to” address.

Minter

The Minter contract is responsible for minting TERM tokens whenever an epoch is flipped. This is done via the updateEpoch function which is called permissionless and can be done once each epoch, whenever Thursday 00:00 is reached. It will then calculate the amount of emissions to all pools, to the RewardsDistributor and to the team and will also distribute these in the same transaction.

As with most VE scopes, the emission is gradually reduced with the exception that it is increased until epochCount becomes 15. More specifically, the weeklyEmissionAmount is initially 10 000 000e18 and is then increased by 3% each epoch until eochCount becomes 15. Afterwards, it decreases by 1% each epoch.

Appendix: weeklyEmission table

Below we will simulate the weeklyEmission variable over the first 20 epochs:

1	10,000,000
2	10,300,000
3	10,609,000
4	10,927,270
5	11,255,088
6	11,592,740
7	11,940,522
8	12,298,737
9	12,667,698
10	13,047,729

11	13,439,160
12	13,842,334
13	14,257,604
14	14,685,332
15	15,125,891
16	14,974,632
17	14,824,885
18	14,676,636
19	14,529,869
20	14,384,570

Appendix: Epoch Flip

Whenever a new week starts on Thursday 00:00, the updateEpoch function can be invoked by anyone which handles the following steps:

a) Update epoch
b) Calculate rebase emissions
c) Calculate team emissions
d) Mint necessary amount
e) Distribute funds to RewardsDistributor, Voter and team

Appendix: Rebase Amount Calculation

Whenever the epoch is updated, the rebase amount which flows towards the RewardsDistributor contract is calculated based on the ratio of the totalSupply and the lockedSupply, specifically based on the following formula:

> ((minted * (totalSupply - lockedLockedSupply)) / totalSupply) * (totalSupply - totalLockedSupply) / totalSupply / 2

Thus, If a large portion of tokens are locked (veTotalSupply is high), then freeFloat is small, and the rebase is small, thus not drastically increasing total supply. If fewer tokens are locked, the rebase is larger.

Appendix: Team Amount Calculation

On top of rebase emissions, the contract also calculates teamEmissions via the computeTeamEmissions function, using the following formula:

> teamRate * (rebase + weeklyEmission) / (10000 - teamRate)

Following this approach, the final teamEmission will be exactly 5% of the overall minted amount.

Core Invariants:

INV 1: updateEpoch is only callable after the TGE

INV 2: updateEpoch must always consume rebase + previousWeeklyEmission + teamEmissions

INV 3: currentEpoch must always point to the TS at the beginning of an epoch

INV 4: updateEpoch must only mint the difference between (rebase + previousWeeklyEmission + teamEmission) - contract balance

INV 5: TERM must be transferred to the RewardsDistributor before checkpointToken is being called

Privileged Functions

- setTeam
- setTeamRate

Issue_59	Manipulation possibility of computeRebase
Severity	Low
Description	As explained in the Appendix, the computeRebase function calculates the rebase amount based on the ratio of lockedSupply to totalSupply. This can be manipulated by minting tokens via the MerkleClaimer contract, for example minting new tokens right before updateEpoch is called, which then in turn increases totalSupply while the lockedSupply still points to the timestamp at the end of the last epoch. This allows to manipulate the rebaseAmount outcome.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Resolved with a refactor of MerkleClaim. The tokens associated with the MerkleClaim are now minted at the start instead of being minted during each claim.

Issue_60	Lack of team address validation can result in breaking updateEpoch
Severity	Low
Description	The setTeam function allows the team to set a new team address. Currently there is no address(0) validation which means in case of a misconfiguration or a compromised team address key, it is possible to set team = address(0). Such a configuration will essentially break the whole architecture indefinitely, as updateEpoch would revert due to a transfer to address(0): term.transfer(address(team), teamEmissions);
Recommendations	Consider validating the setTeam function accordingly.
Comments / Resolution	Resolved by following the recommendation.

Issue_61	Unused return value within updateWeeklyEmission
Severity	**Informational**

Description	The updateWeeklyEmission function exposes the following return value:
	
	return newWeeklyEmission;
	
	This value is however never used:
	
	function updateEpoch() external onlyAfterTGE returns (uint256 epoch) {
	...
	updateWeeklyEmission(previousWeeklyEmission);
	...
	
	}

Recommendations	Consider if this return value should be used, if not consider acknowledging this issue.
Comments / Resolution	Acknowledged.

RewardsDistributor

The RewardsDistributor is the distribution contract for rebase rewards which are allocated to VE token holders. It receives rewards each epoch via the Minter contract in the size of the calculated rebase rewards and distributes these rewards among all veTerm holders.

Appendix: Reward Algorithm

Below we will elaborate the reward algorithm in-depth:

1. Epoch Tracking via lastTokenTime

- Whenever the system updates or checkpoints token balances (e.g., _checkpointToken()), it stores a timestamp in lastTokenTime.
- The contract determines an "epoch" by aligning this lastTokenTime (and subsequent claims) to 1-week boundaries, i.e. (block.timestamp / WEEK) * WEEK.
- Thus, each epoch effectively covers a 1-week span.

2. Proportional Reward Calculation

- For each epoch, the contract records the total amount of newly added tokens via tokensPerWeek[weekCursor].
- During a claim, the user's veNFT (_tokenId) checks how much voting power it had at each weekly checkpoint (timestamp of epoch end), compared to the aggregate total voting power.
- If a veNFT had votingPowerAt at a given block timestamp, while the entire system had totalVotingPowerAt, the fraction (votingPowerAt / totalVotingPowerAt) of that epoch's tokensPerWeek[weekCursor] are claimable:
  > votingPowerAt * tokensPerWeek / totalVotingPowerAt

Claim Steps

- When a user calls claim(_tokenId), the contract loops through each weekly slot from the token's lastClaimOf[_tokenId] up to the most recent checkpoint (lastTokenTime / WEEK) * WEEK.

- At each slot (one-week boundary), it calculates the user's fraction of tokensPerWeek[thatSlot].

- The user's final claim is the sum of these fractions for all unclaimed weeks in that range.

- lastClaimOf is then set to the last updated epoch, which corresponds to the next epoch where rewards can be claimed. If the last updated epoch is 3, rewards are claimable up to epoch 2 and lastClaimOf is set to 3 which then allows for claiming rewards for epoch 3 whenever epoch 4 is reached.

Lock Extension vs. Liquid

- If the user's lock is still active (lockEnd > block.timestamp), the claimed tokens are added ("rebased") into the existing ve lock, increasing the locked amount.

- If the user's lock expired, the claim is transferred directly to the user's address as liquid TERM tokens.

Updating tokenLastBalance

- When new tokens arrive in the contract, _checkpointToken() sees how many more arrived since the previous checkpoint by subtracting tokenLastBalance from the contract's current balance.

- It then updates tokensPerWeek[lastEpoch] with that difference and moves tokenLastBalance upward to match the new total.

- Upon each claim that effectively removes tokens from the contract's balance (whether liquid or by increasing the veNFT lock), the contract subtracts the claimed amount from tokenLastBalance.

Core Invariants:

INV 1: lastTokenTime must always be rounded down by weeks for claimable purposes

INV 2: checkpointToken must only be callable once per epoch

INV 3: claim is only allowed for claims for epochs before lastTokenTime

INV 4: _checkPointToken must distribute rewards always for the past week

INV 5: _lastClaimOf must always be set to weekCursor/Claim.end after a claim

INV 6: rewards can only be claimed if lastTokenTime > weekCursor

INV 7: within _claimable, always the VP at the very end of the week must be considered

INV 8: tokens must have been transferred in before checkPointToken is called

INV 9: lastTokenTime must point to the epoch -1 where last rewards have been distributed

Privileged Functions

- initialize
- tge
- checkpointToken
- setMinter

Issue_62	TERM rewards can always be claimed as liquid tokens
Severity	**High**
Description	When claiming TERM rewards from the RewardsDistributor contract, the contract checks whether the associated VotingEscrow tokenId is expired. If it expires, the TERM rewards are sent as liquid tokens. If it is not expired, the rewards are locked into the token at its existing expiry. However, this logic fails to account for token splitting and merging. When a token is split, its expiry is deleted. Similarly, when a token is merged into another, the merged-from token's expiry is deleted. In both cases, the contract will incorrectly consider the tokenId as expired when its expiration simply isn't recorded in storage anymore. As a result, users can repeatedly merge or split tokens to bypass the intended locking mechanism and claim all TERM rewards as liquid, even if they should be locked. This serves as a mechanism to fundamentally disrupt the business logic intention as these tokens can then be immediately sold on the market.
Recommendations	Consider forcing users to claim rewards on tokens before they are split or merged, and if they do not, any unclaimed rewards on burned tokens should be forfeited. This would eliminate the opportunity to bypass the locking logic.
Comments / Resolution	Resolved by following the recommendation.

Issue_63	Locked rewards in case of zero VP
Severity	**Medium**
Description	Each time updateEpoch() is triggered, the RewardsDistributor.checkpointToken() function is called, recording new tokens that have arrived—even in periods when no veTERM locks exist (i.e., total voting power is zero). These tokens are allocated for the past epoch: *tokensPerWeek[thisWeek - 1 weeks] += toDistribute;* In that case, tokens added to the distributor are allocated to tokensPerWeek[...] for the past, but later claims only award them to veNFT holders if total voting power was non-zero at the relevant weekly boundary. If the system had zero voting power during that epoch, these tokens become effectively “locked” (unclaimable) because the formula: > (votingPowerAt * tokensPerWeek) / totalVotingPowerAt yields zero if totalVotingPowerAt was zero for that epoch. Thus, distributions continue to accumulate in the contract’s balance, but no veNFT can ever claim them retroactively. Additionally, it **may** be a bit more problematic the first time updateEpoch is called. We will now explain why: The initial zero epoch is triggered via the tge function in the TERM contract which invokes Minter.tge and RewardsDistributor.tge, essentially the contract is then in epoch zero. During this call, tokens are minted to the recipient but it's not guaranteed that they will be already added as liquidity. Specifically, that could be a problem if epoch 0 is only very short (e.g. triggered shortly before Thursday 00:00 is reached), which would be logical to prevent other issues from a long zero epoch, such as non existing rewards during epoch zero etc.

	This means essentially that it is possible that no tokens are ever received by contributors during epoch zero and thus no tokenId's can be created during the said epoch. When the next week's boundary is reached, the updateEpoch function is called which mints the first time rewards to be distributed. However, it also mints rebase rewards to the RewardsDistributor (even a large amount due to possible zero lockedSupply which is based on the computeRebase math). These rewards remain essentially locked in the RewardsDistributor and are unclaimable.
Recommendations	In an effort to not disrupt the reward distribution flow with a change, we recommend adding a recover function which allows for withdrawing such stuck funds.
Comments / Resolution	Partially resolved by adding logic that transfers tokens back to the minter in the case of zero VP. This implementation deviates from our recommendation and is thus marked as partially resolved.

Issue_64	Permissionless claim can be used to influence lockedSupply and thus manipulate rebase amount
Severity	**Low**
Description	As we have already elaborated in the Minter contract, the rebaseAmount for each updateEpoch call is influenced by the totalSupply and lockedSupply. We already elaborated one scenario how the totalSupply can be increased via the MerkleClaim contract. Due to the fact that the claim function within the RewardsDistributor is permissionless, a malicious user can claim for the largest tokenIds which then increases totalSupply and thus the user can manipulate rebaseAmount at will.
Recommendations	Consider making the claim functions permissioned if that side-effect is undesired.
Comments / Resolution	Acknowledged.

Issue_65	Full reward allocation even with partial week lock
Severity	Low
Description	Whenever a user creates a tokenId, the checkpoint is updated and reflected with the exact block.timestamp when this tokenId is created. This timestamp is then used within the claimable function and rounded down by weeks: *// Token has never claimed rebase emissions* *// We start from the first checkpoint of the token within the VotingEscrow* *if (weekCursor == 0) {* *(, , uint256 ts,) = IVotingEscrow(ve).tokenPointHistory(_tokenId, 1);* *weekCursor = (ts / WEEK) * WEEK;* *tokenClaim.start = weekCursor;* *}* That means if a tokenId is created during the end of epoch 2, weekCursor will be at the beginning of epoch 2. The epoch end will then be used to determine the VP of that tokenId: *uint256 timestamp = weekCursor + WEEK - 1;* *uint256 votingPowerAt = IVotingEscrow(ve).votingPowerAt(* *_tokenId,* *timestamp* *);* This means, even if a tokenId is minted during the last block of an epoch, this tokenId will still be entitled to the full rewards for the week, even if only existing for one block. A user can thus lock some TERM tokens shortly before Thursday 00:00 and still receive full rewards for last epoch, even if the tokenId has not really been locked for the last week.
Recommendations	Consider acknowledging this issue. This is inherent per design of

	the RewardsDistributor contract.
Comments / Resolution	Acknowledged.

Issue_66	Potential reentrancy attack in case of custom token
Severity	**Informational**
Description	The tokenLastBalance variable tracks how many TERM tokens the contract held after the previous distribution checkpoint. Whenever new TERM arrives, _checkpointToken() compares the new term.balanceOf(address(this)) to tokenLastBalance to see how much more has come in, then updates tokenLastBalance accordingly. This ensures the contract knows exactly how many tokens are fresh and thus to be distributed to VE holders. When users later claim those tokens, the contract subtracts the claimed amounts from tokenLastBalance. Due to the fact that the tokenLastBalance is only decreased after the token transfer (if tokenId is burned or has ended): *if (tokenClaim.amount != 0) {* *(, uint256 lockEnd) = ve.lockedBalances(_tokenId);* *// Token is expired, send the rebase as liquid TERM* *// We use lastOwnerOf here so that recently burned tokens can still claim* *if (lockEnd <= block.timestamp) {* *IERC20(address(term)).transfer(ve.lastOwnerOf(_tokenId), tokenClaim.amount);* *}* *// Token is not expired, add the rebase to the lock* *else {* *IVotingEscrow(ve).increaseAmount(_tokenId,*

	tokenClaim.amount]; } tokenLastBalance -= tokenClaim.amount; It is possible to re enter into Minter.updateEpoch which then calls checkpointToken while the contract balance has been decreased but tokenLastBalance has not decreased, which would effectively result in toDistribute becoming smaller than expected which results in some tokens being effectively locked.
Recommendations	Consider acknowledging this issue since the TERM token does not introduce reentrancy functionalities. If this contract is used with another reward token, this issue must be fixed.
Comments / Resolution	Acknowledged.

Issue_67	claimable does not include unupdated epoch
Severity	**Informational**

Description	The claimable function returns the claimable amount based on lastTokenTime. This does however not mimic the behavior of the claim function:
	
	function claim(uint256 _tokenId) external returns (uint256) {
	// Update the epoch if necessary and trigger the weekly emissions
	*if (IMinter(minter).currentEpoch() < ((block.timestamp / WEEK) * WEEK)) {*
	IMinter(minter).updateEpoch();
	}
	
	As one can see, it is possible that updateEpoch is called which will then distribute rewards and increase lastTokenTime. This is a small discrepancy between the claim and claimable view-only function and can be considered as a wrong return value. It can become problematic for protocols which are built on top of the RewardsDistributor and consider the return value of the claimable function as legitimate.

Recommendations	Consider adding a comment to the claimable function to indicate this specific edge-case.
Comments / Resolution	Resolved by following the recommendation.

Term

The Term contract is a slightly modified ERC20 token contract which extends standard functionality by providing TGE logic and additional mint entry points. The standard mint function is called once per epoch by the Minter contract to mint epoch rewards while the claim function is called by the MerkleClaim contract in an effort to mint tokens which are entitled to users.

Appendix: TGE

The tge function is callable once by the owner and effectively initializes the protocol and epoch 0. An initial amount of 80 million tokens is therefore minted to the recipient address and the tge function on the Minter contract is triggered.

Core Invariants:

INV 1: No tokens can be minted before the TGE has happened

INV 2: Mint is only callable by the Minter contract

INV 3: Claim is only callable by the MerkleClaim contract

Privileged Functions

- tge
- mint
- claim

No issues found.

Governance

GoVeTerm

The GoVeTerm contract is a handler contract which allows for delegating tokenIds to other addresses. This delegation power can then be used for governance decisions within the L2Governor contract.

There are two distinct delegation mechanisms:

a) Full delegation from one address to another address. This allows all tokenIds to be delegated from address A to address B and all tokenIds which are received by address A will be automatically delegated to address B

b) Single token delegation. This automatically delegates a tokenId upon the transfer from one address to another address. This includes edge-cases where the sender has delegated to another address and/or the recipient has delegated to another address.

The contract furthermore implements an upper limit of 128 tokens which can be delegated to a user. This is to combat potential out-of-gas errors due to large loop iterations.

The full delegation mechanism exists out of multiple parts:

a) Proposal: Alice can propose to delegate her tokenIds to Bob

b) Proposal cancel: Alice can revoke her proposal to delegate her tokenIds to Bob

c) Proposal deny: Bob can deny Alice's proposal

d) Proposal acceptance: Bob can accept Alice's proposal and thus Alice delegates all her current as well as future tokenIDs to Bob

e) Proposal renounce: Bob can renounce a delegation by Alice which grants all tokenIds back to Alice

Appendix: Full Delegation Mechanism

Whenever a delegation proposal is accepted, a delegation is renounced or revoked, the _moveAllDelegates function is invoked which will move all tokenId delegations from the delegator to the recipient or vice-versa from the recipient back to the delegator. Important safeguards are present which ensure that MAX_TOKEN_DELEGATIONS is not violated.

Appendix: Token Transfer Delegation

Whenever a veTerm tokenId is transferred from one user to another, this will be reflected in the delegationCheckpoints array, as it will be removed from the original owner and added to the new owner. The function also ensures that delegations in such a scenario are handled. For example if the sender has delegated his tokenIds, it will then be removed from the delegatee and likewise it will be added to the delegatee if the recipient actively delegates.

Appendix: Checkpoint Logic

The delegationCheckpoints logic is heavily inspired by the VotingEscrow algorithm and essentially keeps track of different checkpoints for an address. Whenever a user address receives a delegation of a tokenId or removes a delegation of a tokenId, a new checkpoint with the corresponding new tokenIds and the new timestamp is created. This does not only happen during the _moveAllDelegates function but also during the _updateDelegationForTokenId function. It is important to note that the checkpoint mechanism of the GoVeTerm contract occupies the 0 index while the VE contract starts with index 1 and keeps index 0 unoccupied

Core Invariants:

INV 1: During _moveAllDelegations, the from address must keep all tokenIds which are not owned by the delegator

INV 2: During _moveAllDelegations, even if “to” address has delegated to someone else, received tokenIds must still be delegated to “to” address

INV 3: During _moveAllDelegations, if “to” address has delegated to someone else, need to incorporate owned tokenIds in the expectedLength check

INV 4: RenounceDelegation and revokeDelegation must execute the exact same logic within _moveAllDelegates
INV 5: numCheckpoints must always increase if a new checkpoint for an address is created

INV 6: a tokenId must never be in in two checkpoints during the same block

INV 7: acceptDelegationProposal can only be called if msg.sender is recipient of proposal

INV 8: refuseDelegationProposal can only be called if msg.sender is recipient of proposal

INV 9: cancelDelegationProposal can only be called if proposal is outstanding, by proposer

INV 10: numCheckpoints must increase whenever delegationCheckpoints is changed

INV 11: numCheckpoints always corresponds to the amount of checkpoints from a user

INV 12: within _updateDelegationForTokenId, no duplicate checkpoints per block are allowed

INV 13: within the delegation logic, it is only ever possible to delegate own tokens to someone else. A delegatee cannot delegate his delegations to someone else. It only works if tokens owned

INV 14: Whenever a tokenId is transferred, it must either get added to the recipient's checkpoint or to the recipient's delegatee checkpoint

INV 15: A delegation proposal is not allowed if from already delegated to an address

INV 16: getDelegationCheckpointIndexAt must return the latest checkpoint before/equal lookup TS

INV 17: At the same timestamp, there must never be a tokenId duplicate in any delegationCheckpoint

INV 18: if a new delegationCheckpoint is written in the same block as an update already happened the old checkpoint must be overridden

Privileged Functions

- updateDelegationForTokenId

Issue_68	Incorrect zero index behavior allows counting the same voting power multiple times
Severity	**High**
Description	The GoVeTerm contract calculates historical voting power using the getPastVotes() function. This function performs a binary search with getDelegationCheckpointIndexAt() to find the latest checkpoint a user has before or at a specified timestamp. However, there is an issue in how these two functions interact. Specifically, notice that getDelegationCheckpointIndexAt() uses index 0 as a fallback when no checkpoint exists before the timestamp: *if (delegationCheckpoints[account][0].timestamp > timestamp) {* *return 0;* *}* Even though the earliest checkpoint might be after the target timestamp, index 0 could still hold real voting power from a later timestamp. Since getPastVotes() treats the returned index 0 as valid without checking its timestamp, it can incorrectly use that checkpoint and attribute voting power that didn't exist at the target timestamp, which is because checkpoints start with the zero index during the first creation. For example, suppose Bob initially has no voting power. At time t + 100, Alice transfers a VE token to Bob. If someone calls getPastVotes(Bob, t), the getDelegationCheckpointIndexAt() function finds that Bob's first checkpoint is too recent and returns index 0. But index 0 contains the voting power Bob received at t + 100, so the function incorrectly assumes Bob had that voting power at time t. This allows double-counting: Alice and Bob share the same voting power at time t. A malicious user could repeat this process to share the same voting power with arbitrarily many addresses to

	manipulate governance decisions. The checkpoint implementation seems to be inspired from the _checkpoint logic within the VE contract as it casts different checkpoints with corresponding information. For the VE contract this is the bias/slope/ts/block of a tokenId while for the GoVeTerm contract this is the ts and tokenIds owned by an address. With that in mind, we can also recreate the reason why this issue occurs: The VE contract has no zero index for a corresponding checkpoint: *tokenEpoch = tokenEpoch + 1;* *tokenEpochs[tokenId] = tokenEpoch;* *tokenPointHistory[tokenId][tokenEpoch] = tNewPoint;* (see how tokenEpoch becomes 1 for the first entry) Whereas, within the GoVeTerm contract, the zero index for the checkpoint is actually used: *numCheckpoints[to] = toNum + 1;* *delegationCheckpoints[to][toNum].timestamp = block.timestamp;* (see how toNum is 1 for the first checkpoint) The problem lies within the fact that the important part of the binary search methodology was forked whereas in the VE contract, the zero index is indeed empty while in the GoVeTerm contract, the zero index is not empty (but treated as if) and thus returns a potentially valid entry while it shouldn't.
Recommendations	Consider updating getPastVotes() to handle the scenario where getDelegationCheckpointIndexAt() returns index 0 because no earlier checkpoint exists. One way to do this is to check whether the checkpoint returned has a timestamp after the target timestamp. If it does, return 0 voting power.

Optionally, one can also adjust the whole algorithm to properly push entries starting from the first index instead of the zero index. However, that would require additional validation time and needs to be adjusted in potentially other spots as well, such as here:

```
if (delegationCheckpoints[account][nCheckpoints - 1].timestamp <=
timestamp) {
return (nCheckpoints - 1);
}
```

This should then fetch nCheckpoints instead of nCheckpoints -1. Other spots likely need adjustment as well, such as:

```
        uint32 lower = 0;
        uint32 upper = nCheckpoints - 1;
        while (upper > lower) {
        uint32 center = upper - (upper - lower) / 2; // ceil, avoiding
overflow
        DelegationCheckpoint storage cp =
delegationCheckpoints[account][center];
        if (cp.timestamp == timestamp) {
        return center;
        } else if (cp.timestamp < timestamp) {
        lower = center;
        } else {
        upper = center - 1;
        }
        }
```

Therefore, we recommend choosing the first option, otherwise a full re-audit of the flow is necessary.

Comments / Resolution	Resolved by following the recommendation to check the timestamp of the returned index.

Issue_69	MAX_TOKEN_DELEGATIONS logic can be bypassed to result in an invalid state
Severity	**High**
Description	The GoVeTerm contract sets a MAX_TOKEN_DELEGATIONS limit of 128 to cap how many tokens can be associated with a single address. However, there is an edge case that allows this limit to be exceeded due to how updateDelegationForTokenId() handles certain token transfers when two addresses delegate to the same address. Consider the following scenario involving users A, B, C, and D: - A and B each have 1 token and are delegating to C. - D is delegating 127 tokens to A. The checkpoint for A has now 127 tokenIds while balanceOf remains 1, which brings A's total tokens to 128. - B transfers their token to A which now results in A owning 2 tokens (which are both delegated to C) while having a delegation of 127 tokens This transfer triggers updateDelegationForTokenId(), which checks: *if (from != to && tokenId > 0) {* *// transfer the voting power and check against MAX_TOKEN_DELEGATIONS* *}* Since both A and B are delegating to the same address C, the function sees from == to == C and skips the inner logic. However, this transfer still results in A holding one more token, so it is a mistake not to check A's balance against the MAX_TOKEN_DELEGATIONS. With B's token added, A is associated with 129 tokens (2 in its own balance and 127 delegations), which exceeds the limit. This violation of MAX_TOKEN_DELEGATIONS puts A in an invalid

	state. Any future actions involving delegation or undelegation can revert with TooManyTokenDelegations(). This state may even be permanently unfixable by A, for example, if D also exceeds the MAX_TOKEN_DELEGATIONS, then it would revert if A tried to undelegate D from themselves, so there is nothing A can do to change their delegation state. The functionality thus remains permanently broken.

Recommendations	To ensure that users can't intentionally or unintentionally exceed the MAX_TOKEN_DELEGATIONS and get stuck in an invalid state, consider updating updateDelegationForTokenId() to do the MAX_TOKEN_DELEGATIONS check in all scenarios, even if from == to.
	
	In that scenario, the real token recipient must be checked, which is handled within the following snippet:
	
	// Ensure that the new owner of the token doesn't implicitely go over the limit
	if (owner != to) {
	uint32 ownerNum = numCheckpoints[owner];
	uint256[] storage ownerOld = ownerNum > 0
	? delegationCheckpoints[owner][ownerNum - 1].tokenIds
	: delegationCheckpoints[owner][0].tokenIds;
	
	// Since owner != to we know that owner is delegating to to
	// Hence its own balance is not included in the delegation
	uint256 expectedLengthOwner = ownerOld.length + 1 + ve.balanceOf(owner);
	
	if (expectedLengthOwner > MAX_TOKEN_DELEGATIONS) {
	revert TooManyTokenDelegations();
	}

	} It has to be noted that this issue is non-trivial and potential side-effects from this additional check must be considered. We recommend careful fuzzing based on our provided core invariants. Furthermore. we recommend allocating additional auditing resources to ensure this specific call-path does not introduce side-effects
Comments / Resolution	Resolved by following the recommendation.

Issue_70	Griefing considerations for pending delegations array
Severity	**Low**
Description	In the GoVeTerm contract, when address A wants to delegate to address B, B must explicitly accept (or reject) this delegation. The pending delegation proposals are stored in an array, and the full array may be iterated over in functions like _removeFromPendingDelegations(). Note that since an attacker could propose as many delegations as they want to a specific victim, the gas cost of iterating through the entire array may become very costly and lead to griefing concerns. Fortunately, this is never a permanent problem since users can always repeatedly deal with the first element in the array until the array is empty.
Recommendations	Consider keeping this behavior in mind, and consider documenting the fact that users can always deal with large pending delegation arrays by repeatedly accepting/rejecting the first entry in the array until it is empty.
Comments / Resolution	Resolved by adding comments above the relevant functions describing this behavior.

Issue_71	MAX_TOKEN_DELEGATIONS could be reached with direct VE NFT transfers and createLockFor
Severity	**Low**
Description	The GoVeTerm contract enforces a MAX_TOKEN_DELEGATIONS limit of 128. To prevent spam that could fill up this limit, delegatees explicitly approve each new delegation they receive. However, note that this approval mechanism is bypassed if someone directly transfers a VE NFT to a user or calls VotingEscrow.createLockFor with 1 wei. Since these transfers /mints don't require approval, an attacker could create many NFTs containing dust amounts of TERM and send them to a victim. This could fill the victim's 128 token limit, preventing them from receiving further delegations or creating new VE tokens. Fortunately this would not be a permanent issue, since the victim can always merge or transfer the unwanted NFTs.
Recommendations	Consider documenting this behavior and advising users that if they hit the MAX_TOKEN_DELEGATIONS limit unexpectedly, they should merge or transfer dust NFTs to free up space
Comments / Resolution	Resolved by adding a comment describing this behavior.

Issue_72	Redundant double-entry logic
Severity	**Informational**

Description	The proposal of delegations can either be done via delegate:
	
	function delegate(address delegatee) external {
	proposeDelegation(delegatee);
	}
	
	Or via propose delegation:
	
	function proposeDelegation(address to) public {
	_proposeDelegation(msg.sender, to);
	}
	
	While the delegate function is marked as external, it just calls the public proposeDelegation function. Thus users can also directly call proposeDelegation which will save some gas with virtually no difference. Furthermore such a double entry logic just makes the code unnecessarily complex. On top of that comes the fact that the code also exposes an internal delegate function, which can furthermore increase confusion:
	
	function delegate(address delegator, address delegatee) internal {
	
	if (delegator == address(0)) {
	revert ZeroAddress();
	}
	
	return _delegate(delegator, delegatee);
	}

Recommendations	Consider removing the delegate function.
Comments / Resolution	Resolved by following the recommendation.

Issue_73	delegateWithSig can be griefed
Severity	**Informational**
Description	The delegateWithSig function increments the nonce whenever the signature has been used. If, therefore the function with the same input parameters is called twice, it will revert. Furthermore, it is open for anyone to be called and does not introduce a special caller into the leaf. This can be abused if for example Alice batches multiple transactions outgoing from her multisig, including a delegateWithSig call. Another user can simply frontrun this call which then results in a full revert of the multisig transaction.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

L2Governor

The L2Governor contract is inspired by OpenZeppelin's Governor contract and allows for proposing different proposals by users who have reached the proposalThreshold() with their VP. Once a proposal has been proposed, users can vote either for, against or abstain starting from 15 minutes after the proposal creation (voteStart) until 7 days after proposal creation (voteEnd). With their exact VP at the voteStart timestamp.

The contract incorporates multiple internal functions which are meant to be overridden by the L2GovernorCountingSimple, L2GovernorVotes and L2GovernorVotesQuorumFraction contracts.

This is including but not limited to logic for counting votes, fetching VP, quorum settings and further settings.

The contract furthermore exposes multiple different functions for casting votes:

a) castVote: Simple vote casting

b) castVoteWithReason: Simple vote casting with reason string

c) castVoteWithReasonAndParams: Simple vote casting with reason string and bytes parameters

d) castVoteBySig: Simple vote casting with valid signature

e) castVoteWithReasonAndParamsBySig: Simple vote casting with valid signature with reason string and bytes parameters

Appendix: Proposal States

A proposal can have the following different states:

a) PENDING: After proposing until minute 15

b) ACTIVE: After voteStart +1 until block.timestamp = deadline

c) CANCELED: Once a proposal has been cancelled

d) SUCCEEDED: After block.timestamp > deadline AND reached quorum AND forVotes > againstVotes

e) EXECUTED: After a proposal has been executed

Appendix: Executor Logic

This mechanism is not used within the current scope. Thus we will not explicitly create an appendix.

Core Invariants:

INV 1: duplicate proposals must not be allowed

INV 2: after proposal was made, voting starts 15 minutes + 1 second later

INV 3: after proposal was made, voteEnd is 7 days later

INV 4: during voting, VP from voteStart time is used

INV 5: _countVote can be only called once for specific proposal

INV 6: proposal is considered as pending until block.timestamp > voteStart

INV 7: proposal is considered as active while block.timestamp > voteStart and block.timestamp <= voteEnd

INV 8: proposal quorum is related to voteStart timestamp

INV 9: forVotes + abstainVotes count to quorum

INV 10: executor logic remains unused

INV 11: proposalThreshold must be passed in order to propose

INV 12: a vote must always count the VP of a user at the voteStart timestamp

INV 13: an address can only vote once for a proposalId and then the vote remains immutable

Privileged Functions

- relay

Issue_74	Cancellation of a proposal is not possible
Severity	**Medium**
Description	The contract exposes an internal _cancel function which allows the cancellation of certain proposals. This function can however never be triggered since the corresponding external function is missing. Therefore, even clearly malicious proposals can never be canceled.
Recommendations	Consider adding an external cancel function.
Comments / Resolution	Partially resolved by following the recommendation. It however has to be noted that only the proposer during the PENDING state has the ability to cancel a proposal. This will not allow governance to remove a malicious proposal in emergency situations. We highly recommend adding a governance function for this purpose. Furthermore, the feature of adding a proposer variable to the ProposerCore can result in side-effects which cannot be uncovered during a resolution round as effectively every single state of a proposal needs to be reconsidered.

Issue_75	Vote is immutable
Severity	**Informational**
Description	Once a user has voted for a proposalId, there is no possibility of revoking or changing the vote. Any vote is thus considered as immutable and cannot be reversed. This does only serve as an informational reminder.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_76	castVoteBySig/castVoteWithReasonAndParamsBySig can be griefed
Severity	**Informational**
Description	The castVoteBySig/castVoteWithReasonAndParamsBySig functions call the _countVote function which marks an account as voted, or reverts if an account has already voted. If the function with the same input parameters is called twice, it will revert. Furthermore, it is open for anyone to be called and does not introduce a special caller address into the leaf. This can be abused if for example Alice batches multiple transactions outgoing from her multisig, including a castVoteBySig/castVoteWithReasonAndParamsBySig call. Another user can simply frontrun this call which then results in a full revert of the multisig transaction.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

Issue_77	Missing IERC721Receiver interface check
Severity	**Informational**
Description	The supportsInterface() function has a check against type(IERC1155Receiver).interfaceId, but it does not have a check against type(IERC721Receiver).interfaceId even though it does implement the interface's required functions.
Recommendations	Consider adding a check against type(IERC721Receiver).interfaceId in the supportsInterface() function.
Comments / Resolution	Resolved by following the recommendation.

L2GovernorCountingSimple

The L2GovernorCountingSimple contract is a simple extension of the L2Governor contract which includes vote counting logic.

Core Invariants:

INV 1: Any address can only vote once for a proposal

INV 2: A proposal is succeeded if the forVotes amount is larger than againstVotes amount

INV 3: The quorum is reached if forVotes and abstainVotes become >= quorum

Privileged Functions

- none

No issues found.

L2GovernorVotes

The L2GovernorVotes contract is a simple extension of the L2Governor contract which includes VP fetching logic by consulting the GoVeTerm.getPastVotes function.

Privileged Functions

- none

No issues found.

L2GovernorVotesQuorumFraction

The L2GovernorVotesQuorumFraction contract is a simple extension of the L2Governor contract which includes novel logic for changing the quorum threshold via a checkpoint logic. This ensures that quorum changes do not influence past proposals.

The contract follows a simplified form of checkpoint mechanism for the storage of new quorumNumerator values in an effort to always return the accurate quorumNumerator for a specific block.timestamp. Whenever the updateQuorumNumerator function is called, the new quorumNumerator is pushed into the History.checkpoints[] array with the current block.timestamp.

Privileged Functions

- updateQuorumNumerator

No issues found.

Checkpoint

The Checkpoint contract is a simpler helper library which is used by the L2GovernorVotesQuorumFraction contract in an effort to handle the checkpoint-wise storage of quorumNumerator settings.

Core Invariants:

INV 1: The getAtTimestamp function must always return the nearest checkpoint before or at the provided timestamp

INV 2: The push function must override an existing checkpoint if it has the same block.timestamp as the newly created checkpoint

Privileged Functions

- none

L2GovernorCountingSimple

Issue_78	Unused push operation
Severity	**Informational**
Description	The contract exposes two push operations whereas one is indeed used to create a new checkpoint while the other one remains simply unused: *function push(* *History storage self,* *function(uint256, uint256) view returns (uint256) op,* *uint256 delta* *) internal returns (uint256, uint256) {* *return push(self, op(latest(self), delta));* *}*
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

TerminalGovernor

The TerminalGovernor contract inherits all previously mentioned governance contracts and implements additional functionality related to the proposalNumerator and team settings.

By default, new proposals can only be created if the proposer at least owns 0.2% of the VP

Core Invariants:

INV 1: Only the team address can change the proposalNumerator

INV 2: The proposalThreshold must be determined based on the total VP at block.timestamp -1

Privileged Functions

- setTeam
- setProposalNumerator

No issues found