BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

SwapX Exchange

May 2024

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	SwapX - VE Scope
Website	swapx.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/SwapX-Exchange/contracts-rb/tree/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts https://github.com/SwapX-Exchange/contracts-rb/blob/2900185aa9ec23fa1220b92f555344b5881c19c3/contracts/MonolithicVoter.sol
Resolution 1	https://github.com/SwapX-Exchange/contracts-rb/tree/88fdbc4420e4bd176ef9acd69a1a56827489ef13/contracts
Resolution 2	https://github.com/SwapX-Exchange/contracts-rb/tree/79917a562ef20ea31304073f1737f29aadd6d92a/contracts

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	10	6		4
Medium	11	7		4
Low	11	8		3
Informational	10	3		7
Governance	6			6
Total	48	24		24

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

Global

Issue_01	Lack of safeTransfer usage
Severity	**Informational**
Description	The contract uses the standard transfer pattern for ERC20 transfers, including a true check. This will malfunction for tokens that do not return a boolean on the transfer. This is only rated as informational due to the fact that the SwapX token is used with the transfer pattern.
Recommendations	Consider using safeTransfer.
Comments / Resolution	Acknowledged.

Factories

BribeFactoryV3

The BribeFactoryV3 contract is the factory contract for Bribes. It is responsible for the deployment and configuration of Bribes, which includes the setting of rewardTokens upon each deployment. Bribe deployments (internal & external) are always executed upon a new Gauge deployment and will then be attached to the deployed Gauge.

Furthermore the BribeFactoryV3 serves as the privileged address for Bribes and exposes an interface which allows the execution of the following privileged functions on the Bribes contract:

a) addReward
b) setVoter
c) setMinter
d) emergencyRecoverERC20
e) recoverERC20AndUpdateData

The contract storage holds a defaultRewardToken array which is initially set up with six tokens but can be arbitrarily extended or decreased. This array will be used as initialization value for newly deployed Bribes.

This contract is meant to be used as an implementation contract for a proxy.

Privileged Functions:

- transferOwnership
- renounceOwnership
- createBribe
- setVoter
- setPermissionRegistry
- pushDefaultRewardToken
- removeDefaultRewardToken
- addRewardToBribe
- addRewardsToBribe
- addRewardToBribes

- addRewardsToBribes
- setBribeVoter
- setBribeMinter
- setBribeOwner
- recoverERC20From
- recoverERC20AndUpdateData

Issue_02	Hardcoded defaultRewardTokenAddresses are non existent
Severity	**Informational**
Description	Upon contract initialization, six addresses are pushed into the defaultRewardToken array: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/factories/BribeFactoryV3.sol#L38 Upon the inspection on the block explorer: https://www.okx.com/de/web3/explorer/xlayer/ None of these addresses is corresponding to a token.
Recommendations	Consider removing this redundant operation.
Comments / Resolution	Resolved.

GaugeFactoryV2

The GaugeFactoryV2 contract is the factory for GaugeV2 contracts and is responsible for the correct deployment of these. It is solely meant to be called by the VoterV3 contract and exposes an interface to invoke privileged functions on any deployed GaugeV2 contract, notably:

a) activateEmergencyMode
b) stopEmergencyMode
c) setRewarderPid
d) setGaugeRewarder
e) setDistribution
f) setInternalBribe

The activateEmergencyMode and stopEmergencyMode functions are solely callable by the EmergencyCouncil, which is configured within the PermissionsRegistry contract.

This contract is meant to be used as an implementation contract for a proxy.

Privileged Functions:

- transferOwnership
- renounceOwnership
- setRegistry
- createGaugeV2
- activateEmergencyMode
- stopEmergencyMode
- setRewarderPid
- setGaugeRewarder
- setDistribution
- setInternalBribe

Issue_03	GaugeV2 does not expose setRewarderPid
Severity	**Informational**
Description	The setRewarderPid function allows an allowed address to invoke the setRewarderPid function on a GaugeV2 contract. However, contrary to the GaugeV2_CL contract, the GaugeV2 contract does not expose such a functionality, rendering the function redundant.
Recommendations	Consider simply removing the setRewarderPid function.
Comments / Resolution	Resolved.

GaugeFactoryV2_CL

The GaugeFactoryV2_CL contract is the factory for GaugeV2_CL contracts and is responsible for the correct deployment of these and the corresponding fee vaults (CLFeesVault & CLFeesVault2).
It is solely meant to be called by the VoterV3 contract and exposes an interface to invoke privileged functions on any deployed GaugeV2_CL contract, notably:

a) activateEmergencyMode
b) stopEmergencyMode
c) setRewarderPid
d) setGaugeRewarder
e) setDistribution
f) setInternalBribe
g) setGaugeFeeVault

The activateEmergencyMode and stopEmergencyMode functions are solely callable by the EmergencyCouncil, which is configured within the PermissionsRegistry contract.

This contract is meant to be used as an implementation contract for a proxy.

Privileged Functions:

- transferOwnership
- renounceOwnership
- createGaugeV2
- activateEmergencyMode
- stopEmergencyMode
- setRegistry
- setRewarderPid
- setGaugeRewarder
- setDistribution
- setInternalBribe
- setGaugeFeeVault

No issues found.

Gauges

GaugeV2

The GaugeV2 contract is a simple staking contract that allows users to stake their tokensfor a reward token. It employs similar mechanics to the Synthetix Staking Rewards contract with minor modifications such as an optional rewarder, emergency options and a few other functionalities

The owner of this contract remains the GaugeFactoryV2 contract and can never be changed.

Privileged Functions:

- transferOwnership
- renounceOwnership
- setDistribution
- setGaugeRewarder
- setInternalBribe
- activateEmergencyMode
- stopEmergencyMode
- getReward
- notifyRewardAmount

Issue_04	Governance Privilege: Funds can be permanently locked in the contract
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example: It is possible to add an incompatible ExtraRewarder which then prevents withdrawals.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_05	claimFees will always revert if one of both token is a transfer-tax token
Severity	**High**
Description	The claimFees and corresponding _claimFees function claims the outstanding balance from the Pair/PairFees contract and transfers it to this contract. The return value of the transferred balance is then cached into (claimed0, claimed1) and is then distributed to the internal bribe in the known manner. A problem arises if one of both tokens is a token with a transfer-tax, as that would essentially mean that the cached balance does not correspond to the real received balance and therefore the bribe distribution always reverts. This effectively results in a DoS of the claimFees function and fees being permanently stuck in the pair. This issue is specifically severe because AMMs are/should usually be compatible with fee-on-transfer tokens.
Recommendations	Consider incorporating a before-after balanceOf check and then only distribute as much as was received.
Comments / Resolution	Acknowledged, transfer-tax tokens will not be supported as tokens for liquidity pairs.

Issue_06	Lack of rewarder update during emergencyWithdraw/emergencyWithdrawAmount allows for unfairly stealing majority of rewards from GaugeExtraRewarder
Severity	**Medium**
Description	Rewarders work in a very specific way: They use the latest provided balance of a user to determine rewards for the given time period. **Illustrated:** a) Alice deposits 100e18 tokens at TS = 10_000 b) Alice withdraws 50e18 tokens at TS = 20_000 c) Alice will receive rewards based on 10_000 seconds and 100e18 tokens During the emergencyWithdraw and emergencyWithdrawAmount functions, there is no such onReward call to the Rewarder that adjusts the balance and pays out rewards. The reason for the absence of this call is to ensure that no revert can happen upon the emergency withdrawal. However, this can be abused by depositing a huge amount and then emergency withdrawing this amount. In that scenario, the Rewarder still assumes that the user has a valid stake and will grant rewards even after the withdrawal. *This issue has only been rated as medium severity due to the fact that the emergency mode must be activated.
Recommendations	Consider still executing the onReward call. In case of revert one can simply set the gaugeRewarder to address(0).
Comments / Resolution	Resolved, the onReward call is being executed. In the case of a failure, the gaugeRewarder will be deleted.

	In any scenario where it will exclusively fail for one transaction, the rewarder will be set to address(0) and effectively removed. This may have negative implications if the rewarder is still valid. It may actually be kept as-is because such a scenario will probably never occur. However, a better solution would just to empty the catch and do nothing. **Resolution 2:** The rewarder deletion within the catch block has been removed.

Issue_07	Permanently stuck rewards due to emergencyWithdraw
Severity	**Medium**
Description	The Synthetix reward mechanism increases the rewardPerTokenStored mapping upon every interaction (_deposit, _withdraw, _getReward and notifyRewardAmount . This means that the rewards are distributed on every occasion based on the _totalSupply. **Illustrated:** Alice and Bob have both deposited 100 tokens, with a rewardRate of 1e18 and 100 seconds passed. A third address will deposit now, this will result in rewardPerTokenStored to become 50e18 after 100 seconds, allowing both Alice and Bob to claim 100e18 tokens each. At this point, Alice and Bob's rewards are not updated yet, because the third party deposit will only alter rewardPerTokenStored and not rewards[Alice]/rewards[Bob]. If Alice invokes the emergencyWithdraw function, this will not alter rewardPerTokenStored, but still Bob can only claim 100e18 tokens, as there is no change to Bob's rewards. Since Alice's rewards have not

	been updated beforehand, she cannot claim these after the emergency withdrawal, thus rewards being permanently stuck in the contract. This is a fundamental difference to the masterchef mechanism as within the masterchef algorithm, the reward update is handled differently and lost rewards due to emergency withdrawals are simply allocated amongst the leftover stakers.
Recommendations	Consider incorporating a recover function which allows governance to withdraw these stuck rewards.
Comments / Resolution	Resolved, such a function has been implemented. It is however important to mention that this will not work if the reward token is the stake token.

Issue_08	Small precision can result in loss of rewards
Severity	**Informational**
Description	Currently, rewardPerTokenStored is calculated with a precision of 18 decimals: *(lastTimeRewardApplicable() - lastUpdateTime) * rewardRate * 1e18 / _totalSupply* If the rewardToken is a token with 6 decimals and the TOKEN is a token with 18 decimals, this can round to zero in certain circumstances, preventing the accrual of rewards. *This issue is only rated as informational because SwapX (rewardToken) has 18 decimals.
Recommendations	Consider increasing the precision to 1e24.
Comments / Resolution	Acknowledged.

Issue_09	Contract does not work with 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. This issue has only been rated as informational because this contract is only meant to be used with LPTokens.
Recommendations	Consider not using such tokens.
Comments / Resolution	Acknowledged.

GaugeV2_CL

The GaugeV2_CL contract is a simple staking contract that allows users to stake tokens for a reward token. It employs similar mechanics to the Synthetix Staking Rewards contract with minor modifications such as an optional rewarder, emergency options and a few other functionalities

The owner of this contract remains the GaugeFactoryV2_CL contract and can never be changed.

Notably, the IRewarder interface is not corresponding to the GaugeExtraRewarder within this scope but rather to another rewarder implementation.

Privileged Functions:

- transferOwnership
- renounceOwnership
- setDistribution
- setGaugeRewarder
- setFeeVaults
- setRewarderPid
- setInternalBribe
- activateEmergencyMode
- stopEmergencyMode
- getReward
- notifyRewardAmount

Issue_10	Governance Privilege: Funds can be permanently locked in the contract
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example: It is possible to add an incompatible ExtraRewarder which then prevents withdrawals.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_11	Incorrect order of operations will dilute rewards in extraRewarder
Severity	**High**
Description	Currently, during the _deposit function, tokens are being transferred into the contract before the onReward function on the extraRewarder is invoked. Since the current balance of the Gauge is used as lpSupply divisor in the ExtraRewarder, this will dilute rewards because it updates the pool state with the new amount already included in the balance, while in fact it should update the pool state with the historic balance.
Recommendations	Consider executing the onReward call before the transfer.
Comments / Resolution	Resolved.

Issue_12	claimFees will always revert if one of both token is a transfer-tax token
Severity	**High**
Description	The claimFees and corresponding _claimFees function claims the outstanding balance from the CLFeesVault contract and transfers it to this contract. The return value of the transferred balance is then cached into (claimed0, claimed1) and is then distributed to the internal bribe in the known manner. A problem arises if one of both tokens is a token with a transfer-tax, as that would essentially mean that the cached balance does not correspond to the real received balance and therefore the bribe distribution always reverts. This effectively results in a DoS of the claimFees function. This issue is specifically severe because AMMs are/should usually be compatible with fee-on-transfer tokens.
Recommendations	Consider incorporating a before-after balanceOf check and then only distribute as much as was received.
Comments / Resolution	Acknowledged.

Issue_13	Lack of rewarder update during emergencyWithdraw/emergencyWithdrawAmount allows for unfairly stealing majority of rewards from GaugeExtraRewarder
Severity	**Medium**
Description	ExtraRewarders work in a very specific way: They use the latest provided balance of a user to determine rewards for the given time period. **Illustrated:** a) Alice deposits 100e18 tokens at TS = 10_000 b) Alice withdraws 50e18 tokens at TS = 20_000 c) Alice will receive rewards based on 10_000 seconds and 100e18 tokens During the emergencyWithdraw and emergencyWithdrawAmount functions, there is no such onReward call to the ExtraRewarder that adjusts the balance and pays out rewards. The reason for the absence of this call is to ensure that no revert can happen upon the emergency withdrawal. However, this can be abused to deposit a huge amount and then emergency withdraw this amount. In that scenario, the ExtraRewarder still assumes that the user has a valid stake and will grant rewards even after the withdrawal. *This issue has only been rated as medium severity due to the fact that the emergency mode must be activated.
Recommendations	Consider still executing the onReward call. In case of revert one can simply set the gaugeRewarder to address(0).
Comments / Resolution	Resolved, it seems that the recommendation was misunderstood here. The idea was that governance will set the rewarder to address(0)

	manually in such a scenario. Consider removing this delete call. **Resolution 2:** The delete call has been removed. Governance will manually set the gaugeRewarder to address(0).

Issue_14	Permanently stuck rewards due to emergencyWithdraw
Severity	**Medium**
Description	The Synthetix reward mechanism increases the rewardPerTokenStored mapping upon every interaction (_deposit, _withdraw, _getReward and notifyRewardAmount). This means that the rewards are distributed on every occasion based on the _totalSupply. **Illustration:** Alice and Bob have both deposited 100 tokens, with a rewardRate of 1e18 and 100 seconds passed. A third address will deposit now, this will result in rewardPerTokenStored to become 50e18 after 100 seconds, allowing both Alice and Bob to claim 100e18 tokens each. At this point, Alice and Bob's rewards are not updated yet, because the third party deposit will only alter rewardPerTokenStored and not rewards[Alice]/rewards[Bob]. If Alice invokes the emergencyWithdraw function, this will not alter rewardPerTokenStored, but still Bob can only claim 100e18 tokens, as there is no change to Bob's rewards. Since Alice's rewards have not been updated beforehand, she cannot claim these after the emergency withdrawal, thus rewards being permanently stuck in the contract.

	This is a fundamental difference to the masterchef mechanism as within the masterchef algorithm, the reward update is handled differently and lost rewards due to emergency withdrawals are simply allocated amongst the leftover stakers.
Recommendations	Consider incorporating a recover function which allows governance to withdraw these stuck funds.
Comments / Resolution	Resolved, such a function has been implemented. It is however important to mention that this will not work if the reward token is the stake token.

Issue_15	Small precision can result in loss of rewards
Severity	**Informational**
Description	Currently, rewardPerTokenStored is calculated with a precision of 18 decimals: *(lastTimeRewardApplicable() - lastUpdateTime) * rewardRate * 1e18 / _totalSupply* If the rewardToken is a token with 6 decimals and the TOKEN is a token with 18 decimals, this can round to zero in certain circumstances, preventing the accrual of rewards. *This issue is only rated as informational because SwapX (rewardToken) has 18 decimals.
Recommendations	Consider increasing the precision to 1e24.
Comments / Resolution	Acknowledged.

Issue_16	Contract does not work with 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. This issue has only been rated as informational because this contract is only meant to be used with non-transfer tax tokens.
Recommendations	Consider not using such tokens.
Comments / Resolution	Acknowledged,

Issue_17	GaugeRewarder cannot be set back to address(0) once set
Severity	**Low**
Description	The setGaugeRewarder function allows the contract owner to set a corresponding rewarder. However, once this variable is set, it cannot be set back to address(0), which will then further disturb the business logic because at some point a dummy contract needs to be deployed and set.
Recommendations	Consider allowing the setting of gaugeRewarder back to address(0).
Comments / Resolution	Resolved.

GaugeExtraRewarder

The GaugeExtraRewarder contract is a simple rewarder contract which is meant to be employed on top of staking contracts with a matching interface. Its sole purpose is to distribute an additional reward token on top of the standard staking protocol and it incorporates the standard Masterchef reward algorithm for that purpose

Contrary to the traditional ExtraRewarder, this contract employs a setDistributionRate function which allows the owner to set a rewardRate for a determined period, which is **one week**.

Illustrated this means if the setDistributionRate function is invoked on TS = 1716422400 with amount = 604800e18, this will distribute 1e18 tokens per second for one week with the lastDistributedTime being set to 1717027200. Once the lastDistributedTime is exceeded, no further rewards are distributed until the owner again invokes the setDistributionRate function.

Tokens are meant to be transferred directly to the contract **before** the setDistributionRate function has been called.

Privileged Functions:

- transferOwnership
- renounceOwnership
- setDistributionRate
- recoverERC20

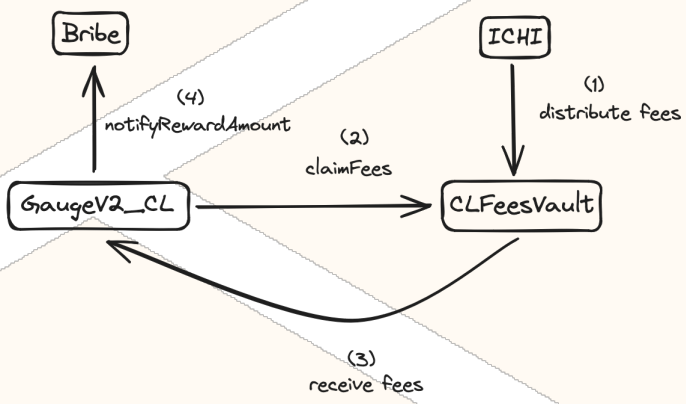
Issue_18	Reentrancy vulnerability allows for draining rewards if rewardToken is ERC777 token or token with a hook
Severity	**High**
Description	The onReward function allows for distributing reward tokens to a recipient. This is done in such a manner that the rewardToken is transferred out **before** the new rewardDebt is set. This wrong order of operations allows a malicious user to reenter into the gauge's original deposit/withdraw/getReward/whatever function (if it is not guarded) and then trigger the onReward function again, effectively withdrawing the same amount of rewards over and over again.
Recommendations	Consider using a nonReentrant modifier
Comments / Resolution	Resolved. However, it was not resolved as per our recommendation. The used implementation will not work for tokens that revert upon zero-transfers. Consider implementing an if condition and only execute the transfer if the pending amount is in fact non-zero

Issue_19	Low precision will result in loss of rewards for reward tokens with 6 decimals
Severity	**High**
Description	Currently, the accRewardPerShare is calculated with a precision of 12 decimals: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/GaugeExtraRewarder.sol#L165 If the rewardToken has 6 decimals and the stakingToken has 18 decimals, this will not work: accRewardPerShare = accRewardPerShare + **(reward * (ACC_TOKEN_PRECISION) / lpSupply);** 100e6 * 1e12 / 1 000 000e18 = 0 *This issue is rated as high severity because the low precision of 12 decimals will render rewards for tokens with 6 decimals completely unusable.
Recommendations	Consider using 1e24 as a precision factor.
Comments / Resolution	Acknowledged.

Issue_20	Recovery of tokens not possible if *block.timestamp >= lastDistributedTime*
Severity	**Low**
Description	Primary, the recoverERC20 function is responsible for recovering tokens that have been allocated as reward tokens, as this will decrease rewardPerSecond accordingly. However, there is also the possibility to recover tokens which have been sent to the contract by accident (including the rewardToken). This will only work as long as the reward epoch has not been exceeded, otherwise the function will revert due to an underflow: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/GaugeExtraRewarder.sol#L184 which will essentially prevent to rescue rewardTokens which have not been allocated as reward but rather were received by a donation.
Recommendations	Consider implementing a special condition for that scenario which still allows to recover the rewardToken. Optionally it is also possible to just call setDistributionRate with amount = 0, which then sets rewardPerSecond to zero and therefore circumvents the above mentioned condition.
Comments / Resolution	Acknowledged.

CLFeesVault

The CLFeesVault contract is a simple storage contract that receives token0/token1 from the corresponding ICHI implementation. These fees will then be claimed by the Gauge contract and further distributed to the corresponding Bribe contract.



Privileged Functions:

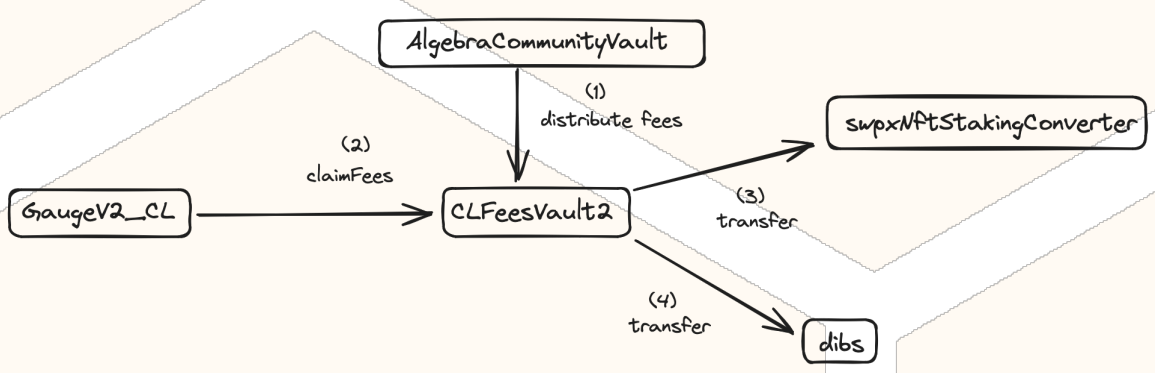
- setVoter
- setPermissionRegistry
- setPool
- emergencyRecoverERC20

No issues found.

CLFeesVault2

The CLFeesVault2 contract is a simple storage contract that receives token0/token1 from the AlgebraCommunityVault. These fees will then be claimed by the Gauge contract and further distributed to:

a) Dibs (83.3%)
b) swpxNftStakingConverter (16.7%)



Privileged Functions:

- setDibs
- setNftStaking
- setPairFactory
- setVoter
- setPermissionRegistry
- setPool
- emergencyRecoverERC20
- claimFees

No issues found.

Core

VotingEscrow

Disclaimer: The checkpoint and delegation algorithm is not included in the audit scope. It is expected that these work flawlessly and are robust against manipulation.

The VotingEscrow contract is the heart of the VE implementation. It allows users to lock their SwapX tokens for a specified amount of time in exchange for a NFT that represents voting power. The higher the balance and the higher the lock duration, the higher the voting power. The VE contract allows users a variety of interactions:

a) deposit_for: Add value to an existing tokenId (no limitation)
b) create_lock: Lock tokens for a specific amount of time and receive a tokenId as receipt which reflects the VP
c) create_lock_for: Similar to create_lock but allows for an arbitrary recipient of the tokenId
d) increase_amount: Add value to an existing tokenId (must be approved for this tokenId or owner)
e) increase_unlock_time: Increase the time when the tokenId is unlocked. This will increase the VP (must be approved for this tokenId or owner)
f) withdraw: Allows to withdraw an unlocked tokenId
g) merge: Allows to merge two owned or approved tokenIds
h) split: Allows to split an approved or owned tokenId to multiple new tokenIds

Once users have received their tokenIds, they will automatically receive rewards from the RewardsDistributor contract and are additionally able to vote for gauges where votes will become eligible for additional rewards (bribes).

Additionally, the contract offers a delegation mechanism which allows to delegate the VP to an arbitrary address. However, this logic was never used in the past and is also not used in this architecture. It is also **not deemed** as bug-free. Similar to the delegation mechanism, are the view functions not bug-free, which allow to fetch VP or the supply at a specific block.number, as this can become inaccurate due to the extrapolation approach. This logic however also remains unused.

Appendix: Checkpoint Algorithm

The checkpoint logic is inspired from Curve's VE implementation and essentially decays a tokenId's VP over time. 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) 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:

user_point_history[epoch]

- bias: slope * (lockedEndTs - currentTs)
- slope: amountToLock / 4 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)*

Which simply decays the bias over time, while using the initial bias and currentTs, indicating a decreased VP value over time.

b) The totalSupply of all tokenIds. This is facilitated in:

point_history[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.

Out of scope changes:

- **Implementation of cross-contract claim call to RewardDistributor**

Privileged Functions:

- setTeam
- setArtProxy
- setVoter
- voting
- abstain
- attach
- detach

Issue_21	Governance Privilege: Storage control
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_22	Burning tokenId will permanently lock reward tokens in RewardsDistributor
Severity	**High**
Description	The RewardsDistributor contract distributes rewards for tokenIds based on their VP and the overall VP, each epoch. Rewards can therefore only be fetched for the corresponding tokenId and will either be transferred to the current owner or added towards the tokenId: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/RewardsDistributor.sol#L292 If a tokenId is burned without the corresponding rewards being claimed beforehand, these rewards are essentially stuck forever in the RewardsDistributor contract. This issue is also present for merge and split.

Recommendations	An isolated fix would be to establish an interconnection between the VotingEscrow and the RewardsDistributor that automatically claims all rewards before the tokenId is burned. However, due to the fact that this codebase is widely used and battle-tested, and this change is quite intrusive, it is important to weigh the issue severity vs security benefits of a battle-tested codebase. Therefore we come to the conclusion that a simple frontend notice which raises awareness for users to claim rewards shall be sufficient
Comments / Resolution	**Failed resolution.** Within the split/withdraw/merge functions, a claim call to the RewardsDistributor is invoked. While we have specifically mentioned that we do not recommend such a fix, it still has been implemented. During our validation it was investigated that this will expose a problem because the withdraw call will revert if *block.timestamp = _locked.end.* The reason for this is because at this timestamp, it will invoke the deposit_for function which will then revert due to the reentrancy check: https://github.com/SwapX-Exchange/contracts-rb/blob/88fdbc4420e4bd176ef9acd69a1a56827489ef13/contracts/RewardsDistributor.sol#L296 This edge-case highlights once again the mandatory carefulness when implementing changes. Therefore we recommend sticking to our previous recommendation to reverse this change and stick to a frontend notice. This moreover just highlights the need for sufficient testing, as this issue would most probably have been caught by testing edge-cases.

	Resolution 2: This call has been removed. It is advised to notify users on the frontend that they should manually claim any unclaimed rewards.

Issue_23	Merge allows to bypass expiry safeguards
Severity	**Low**
Description	The merge function allows to merge two tokenIds into one tokenId. More specifically, it allows for burning one tokenId and adding the value to another tokenId. When doing this, the larger of both unlockTimes is used, which prevents a trick to withdraw a tokenId earlier than the unlockTime. Upon careful inspection of the codebase, one realizes that it is not allowed to further extend an expired tokenId, nor add value to it: a) https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VotingEscrow.sol#L778 b) https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VotingEscrow.sol#L829 c) https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VotingEscrow.sol#L844 This safeguard can be bypassed by using the merge function, which now simply allows to add an expired lock to a non-expired lock or to

	increase the value of an expired lock. We could however not determine any negative side-effects from that behavior.
Recommendations	An isolated fix would be to ensure that both tokenIds (_from and _to) are not expired. However, due to the fact that this codebase is widely used and battle-tested, and this change is quite intrusive, it is important to weigh the issue severity vs security benefits of a battle-tested codebase. Therefore we come to the conclusion that a simple frontend notice which raises awareness for users to claim rewards shall be sufficient
Comments / Resolution	Acknowledged.

Issue_24	Incorrect supply update during merge will falsify supply
Severity	**Low**
Description	Whenever two tokenIds are merged, the tokenId "_from" is reset and burned. The value of this tokenId is then attached to tokenId "_to", which increases the overall supply: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VotingEscrow.sol#L731 However, the supply is not decreased beforehand within the merge function, which will effectively incorrectly inflate the supply everytime a merge happens. Fortunately, this issue does not have an impact as the supply variable

	is not actively used for business logic purposes, hence this issue is only rated as a low severity.
Recommendations	Consider decreasing the supply before _checkpoint and _burn is invoked.
Comments / Resolution	Resolved.

Issue_25	Unsafe casting to int128 can result in loss for users
Severity	**Low**
Description	Within the codebase there are several unsafe castings to int128, notably during the deposit interaction towards a position: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VotingEscrow.sol#L735 This will not work for amounts which are larger than int128. However, in this architecture this issue can be safely ignored because the SWPX token will never have such a large supply.
Recommendations	Consider acknowledging this issue.
Comments / Resolution	Acknowledged.

VoterV3

The VoterV3 contract is the entry contract which orchestrates the voting mechanism. Whenever a new epoch has started, which is each Thursday 00:00 UTC, users can vote with their tokenId for one or more gauges. Voting will have the following benefits:

a) Allocates rewards to gauges based on the % of the overall allocated VP
b) Receive rewards in form of swap fees and external bribes

Votes in epoch x are casted for epoch x and once epoch x has surpassed (epoch x+1 is initiated), the rewards will be allocated accordingly to gauges based on the VP weights. Users can vote anytime during an epoch and are also able to abstain their votes which have already been made.

To facilitate same votes for subsequent epochs the contract exposes a poke function which simply uses the previous vote configuration for subsequent epochs.

Contrary to Thena's implementation, only privileged addresses can create new gauges which greatly limits the existence of low quality gauges and reduces the risk of the distributeAll function to run out of gas.

Existing gauges can be killed and revived which means they can be excluded from voting or included again, per desire of governance.

Whenever an epoch has been surpassed, the next epoch update is facilitated in the Minter contract which then invokes the notifyRewardAmount function to initiate the reward distribution towards the different gauges.

Privileged Functions:

- transferOwnership
- renounceOwnership
- setVoteDelay
- setMinter
- setBribeFactory
- setPairFactory

- setPermissionsRegistry
- setNewBribes
- setInternalBribeFor
- setExternalBribeFor
- addFactory
- replaceFactory
- removeFactory
- whitelist
- blacklist
- killGauge
- reviveGauge
- createGauges
- createGauge

Out of scope changes:

- **Refactoring of the isAlive logic (includes several implications on the contract such as during _updateForAfterDistribution)**
- **removal of whitelist/blacklist mechanism**

As per our guidelines, it is strongly discouraged to make out of scope changes. This is underlined by the error which has been introduced within the VotingEscrow contract for the claim call during the “withdraw” function. These changes should be reversed and the contract shall be submitted for revalidation.

Issue_26	Governance Privilege: Storage manipulation
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. For example, Bribes can be changed which will then prevent any _reset call, thus resulting in permanently used tokenIds, or the gaugeFactory can be changed which means the approval of a newly created gauge could be done to a malicious gauge. Furthermore, it is used under a proxy contract.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_27	Flaw within _reset incorrectly reduces allocation in bribe
Severity	**High**
Description	Whenever the _reset function is invoked, this will withdraw the corresponding amount from both Bribe contracts: *IBribe(internal_bribes[gauges[_pool]])._withdraw(uint256(_votes[i]), _tokenId);* *IBribe(external_bribes[gauges[_pool]])._withdraw(uint256(_votes[i]), _tokenId);* The problem hereby is that this also happens in the scenario where the tokenId has no allocation in the current epoch. This will result in two problems: **Issue A:** Users can abuse this flaw to steal rewards from the gauge by first voting with a tokenId with a higher allocation and then resetting the previous tokenId which was deposited one epoch before. This allows to then again abandon the vote with the more valuable tokenId without reducing the allocation due to withdraw amount being larger than the balance: *if (amount <= _balances[_owner][_startTimestamp])* **Illustrated:** 1. Alice votes with tokenId = 1 with a low value (1 WEI) during epoch 10 2. Alice votes with tokenId = 2 with a large value (100_000e18) during epoch 11, this will set the balance on the Bribes contracts to 100_000e18:

_balances[_owner][_startTimestamp] = _lastBalance + amount;

3. Alice calls reset with tokenId = 1, due to the blunder, it will decrease the balance in the Bribes contract to 100_000e18 -1:

IBribe(internal_bribes[gauges[_pool]])._withdraw(uint256(_votes[i]), _tokenId);

4. Alice now calls reset with tokenId = 2, the tokenId is now free'd up but the balance in the Bribes contract is not decreased because in step 3, the balance has been decreased by 1 we and the if-clause is not triggered:

if (amount <= _balances[_owner][_startTimestamp])

Issue B:

A user's allocation will be unlawfully reduced if a tokenId that has a previous allocation is withdrawn or used for another gauge voting.

Illustrated:

1. Alice votes with tokenId = 1 which is worth 100e18 for Gauge SWPX/ETH (epoch 10)

2. Alice votes with tokenId = 2 which is worth 110e18 for Gauge SWPX/ETH (epoch 11)

-> she will receive a balance of 110e18 on the corresponding bribes

3. Alice votes with tokenId = 1 which is worth 100e18 for Gauge SWPX/USDC (epoch 11)

-> due to the _reset call, it will unlawfully reduce the allocation which

	Alice has gained in the previous step, effectively reducing her balance to 10e18.
Recommendations	Consider not withdrawing an allocation from a gauge if a tokenId has not yet voted in the current epoch. This can be trivially done by wrapping the withdraw call into this if-clause: *if(lastVoted[_tokenId] > _time)*
Comments / Resolution	Resolved.

Issue_28	Loss of distributed rewards if no votes occurred during one epoch
Severity	**High**
Description	Whenever a new epoch has started, rewards for the last epoch's votes are distributed via the Minter which invokes notifyRewardAmount: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L686 This will then increase the index using the reward per weight scheme: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L695 The problem: If no votes have occurred during the past epoch, this means the index will not be increased thus no rewards are allocated. However, they are still being transferred into the contract: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L688 Which now results in these tokens being stuck.
Recommendations	Consider simply incorporating a recoverERC20 function that allows for recovering stuck funds. This is also helpful with regards to another issue.
Comments / Resolution	Resolved.

Issue_29	Kill and revive of gauge in same epoch will break accounting
Severity	**High**
Description	Consider the scenario where we are in a running epoch and votes have already been casted to different gauges, this will have the following storage impact: *weightsPerEpoch[time][gauge] = VP* *votes[voter][gauge] = VP* *totalWeightsPerEpoch[time] = VPAggregated* Most importantly is the totalWeightsPerEpoch mapping, as this will be used for the index calculation within notifyRewardAmount when a new epoch is introduced: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L691 If a gauge is now killed, the totalWeightsPerEpoch mapping is decreased with the corresponding VP which was allocated to this gauge during the current epoch: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L327 This is correct because this gauge should not get an allocation. Several problems will now arise if a gauge is revived in the same epoch, because this will not increase the totalWeightsPerEpoch mapping back to the old value but the individual gauge weight is still existent (weightsPerEpoch)

	1) Upon the distribution (after the epoch update), this means that the gauge will still get an allocation (due to the still existing weightsPerEpoch mapping) but the index variable was not correctly adjusted (due to the missing increase of totalWeightsPerEpoch when the gauge is revived). Therefore, the index is larger as expected and tries to distribute more rewards than received. 2) if users _reset their votes (in the same epoch), this will decrease the totalWeightsPerEpoch mapping by the amount of VP that was allocated to the gauge: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L393 This is wrong because it was already decreased due to the kill interaction. This could even prevent _reset due to an underflow revert. 3) If the gauge will be revived and killed again, this will decrease totalWeightsPerEpoch[time] twice.
Recommendations	Consider not allowing to kill and revive a gauge in the same epoch. A simple mapping *hasGaugeKilled[gauge] = epoch* can be used for that purpose.
Comments / Resolution	Failed resolution, the isAlive logic has been completely refactored without us recommending it, this introduces redundant complexity which can be avoided. We recommend to do the following steps: a) Reverse all changes corresponding to the isAlive logic b) Simply use a hasGaugeKilled[gauge] = epoch mapping which is set whenever the gauge is killed and is used during the reviveGauge function as safeguard.

	This recommendation is not optional but mandatory since the isAlive logic is also used in further places. **Resolution 2:** The recommended fix has been introduced.

Issue_30	Accounting will be broken if not all gauges are updated
Severity	**Medium**
Description	Whenever rewards are provided via the notifyRewardsAmount function, this will increase the index variable using the token per weight approach: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L697 This essentially means indexIncrease * totalWeight = providedReward Once this has happened, the next step is to invoke distribute/distributeAll, which allocates the calculated rewards based on the overall votes and each gauges votes: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L764 The calculation for this is trivially done by multiplying the gauges votes with the index, as this will now yield how much tokens will be allocated to each specific gauge: https://github.com/SwapX-Exchange/contracts-

rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L793

The problem with this flow is the assumption that each gauge is updated after an epoch has passed, this is however not guaranteed as there is simply no check in the contract which ensures this.

Therefore, if a gauge is not updated, this will not set supplyIndex[gauge] to the most recent index and therefore the share calculation during the update in the next round is flawed.

Consider the following PoC:

1) There are currently two gauges:

WETH/USDC
WBTC/USDC

both gauges have received a VP of 50

2) update_period is invoked which then updates the index based on the totalWeight and the amount of rewards:

https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L697

For simplicity reasons let's just consider that there are 100e18 reward tokens and a totalWeight of 100e18, this will set index to 1e18

Therefore, if _distribute is invoked, both gauges would receive 50e18 tokens:

https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/Vo

terV3.sol#L793

and supplyIndex[gauge] is updated afterwards

3) Now the WBTC/USDC gauge is not distributed within this epoch and therefore the supplyIndex of this gauge is not updated, it is still zero.

4) For the next period, we have 100e18 reward tokens and solely the WBTC/USDC vault got an allocation of 100e18 VP. Therefore, index is set to 1e18+1e18:

https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L697

5) The distribute function is now invoked for the WBTC/USDC vault:

https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L793

remember, how the gauge got an allocation of 100e18 for the second period and how the index = 2e18 but supplyIndex for this gauge is zero (due to the fact that it was not updated in the last epoch).

This will effectively attempt to distribute 200e18 tokens to the WBTC/USDC gauge, while the gauge should effectively only receive 50e18 (epoch 1) and 100e18 (epoch 2).

This will break the whole mechanism.

*This issue is rated as medium severity only because only privileged addresses can create new gauges.

Recommendations	Due to the fact that only privileged addresses can create new gauges, the risk of this issue happening is greatly limited. Additionally we recommend keeping an off-chain system which ensures that no gauge remains undistributed (similar to what Thena does).
Comments / Resolution	Acknowledged.

Issue_31	Killing gauge after index increase will result in stuck funds
Severity	**Medium**
Description	Whenever a new epoch is introduced, this will invoke the notifyRewardAmount function that then increases the index. This flow was already explained multiple times during this report. It is a valid scenario that not immediately all gauges are “distributed” after the index has increased. Specifically, it is possible to kill a gauge after the index has increased. If now a gauge is successfully killed, this means that isAlive[gauge] is set to false. This has the following impact whenever the gauge is now updated: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VoterV3.sol#L794 claimable[gauge] is not increased, thus no rewards will be sent to the gauge. This is totally fine as the gauge was killed and should not receive any rewards. However, these rewards simply remain stuck permanently in the VoterV3 contract.

Recommendations	Consider simply incorporating a recoverERC20 function that allows for recovering stuck funds.
Comments / Resolution	Acknowledged.

Issue_32	Inconsistency in Masterchef interaction
Severity	**Low**
Description	Upon the distributeAll function, an external updatePool call to the Masterchef is executed. This call is however not executed upon any other function. In the worst case scenario, an epoch is updated using the distribute function instead of the distributeAll function, which would then not update the Masterchef.
Recommendations	The Masterchef is not in the current scope but will be part of the next iteration, therefore we cannot fully determine the impact but still recommend to execute the updatePool call on both distribute functions as well.
Comments / Resolution	Resolved.

Issue_33	replaceFactory function is flawed
Severity	**Low**
Description	The replaceFactory function is used to add a new gauge and pair factory to the position of an existing gauge and pair factory. This is simply done by replacing it. The validation is however incorrect, as it ensures that the newly added factories are already existent, which is wrong: *require(isFactory[_pairFactory], '!fact');* *require(isGaugeFactory[_gaugeFactory], '!gFact');* Thus this function will never work.
Recommendations	Consider inverting the checks.
Comments / Resolution	Resolved.

Issue_34	Change of internal_bribe assignment may leave internal_bribe in corresponding gauge unchanged
Severity	**Informational**
Description	Governance of this contract can change the internal_bribe assignment for a gauge which means that vote deposits and withdrawals will be forwarded to a different gauge than initially configured. The problem hereby is that each gauge will allocate swap fees to the internal_bribe in the Gauge's storage. This can easily result in an inconsistency where the Gauge's internal_bribe remains unupdated and keeps distributing rewards to the old Bribes contract instead of the new one.
Recommendations	Consider keeping this fact in mind and manually update the gauge's internal_bribe as well.
Comments / Resolution	Acknowledged.

Issue_35	Contract is not compatible with 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. *This issue is only rated as informational because the SwapX token has no transfer-tax.
Recommendations	Consider not using these tokens.
Comments / Resolution	Acknowledged.

MinterUpgradeable

The MinterUpgradeable contract handles the emission calculation and distribution of the SwapX token. Whenever a new epoch is started, tokens are distributed in the following manner:

a) The initial distribution of tokens per epoch is 2_000_000 SwapX token, which is decreased by 1% every epoch
b) Between 10% and 30% goes to the RewardDistributor, this will start with 1% and increases every epoch by 10 BPS
c) Between 3% to 5% goes to the team, whereas in the first 12 weeks this will distributed to the Masterchef
d) Between 5% to 10% goes to the referralAddress
e) The leftover will be distributed towards all Gauges, which is between 91% and 82% of the overall weekly emissions.

It is important to mention that the Minter is the only contract that can update epochs and all other contracts in the architecture (VoterV3, GaugeV2, Bribes, RewardsDistributor) are only following the epoch update of the Minter contract. Epoch updates are permissionless but for ideal execution the update_period should be triggered via VoterV3.distributeAll at the very first block whenever a new epoch has started, which is always Thursday 00:00 UTC.

Privileged Functions:

- transferOwnership
- renounceOwnership
- startEpoch
- setTeam
- acceptTeam
- setVoter
- setTeamRate
- setEmission
- setReferralRate
- setReferralAddress
- setRewardDistributor

Issue_36	Governance Privilege: Governance can steal all minting rewards
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. This includes setting the voter variable to any address which then allows for stealing rewards which are meant to be distributed to gauges.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_37	Emissions can be changed in hindsight via setEmission function
Severity	**Medium**
Description	The setEmission function allows for altering the EMISSION parameter. This can be done while an epoch update is outstanding which will essentially alter the reward emission for the outstanding epoch as well and not only for future epochs.
Recommendations	Consider invoking update_period before the EMISSION variable is adjusted.
Comments / Resolution	Resolved.

Bribes

The Bribe contract is a simple distribution contract which is inherently connected to the VoterV3 contract. Each Bribe contract is linked to a specific gauge and whenever users vote for this gauge, they will gain an allocation in the Bribe contract.

The contract owner can add one or more tokens as reward tokens which then allows anyone to deposit these into the Bribe contract. The idea behind this scheme is to incentivize votes for a specific Gauge. A famous example is the Overnight protocol, which regularly bribes their pools in an effort to increase the votes towards their pools.

The whole mechanism is epoch based, which means when users vote during epoch 1, their allocation will be assigned towards epoch 2 and claimable once epoch 2 has been surpassed.

As the reward mechanism, this contract uses a trivial RPT (reward per token) algorithm which then distributes the bribed rewards on a pro-rata base of the overall assigned VP for each specific Bribe contract.

There are several mechanisms to claim rewards, either directly via the getReward function or via the VoterV3 contract as an interface. The latter mechanism is just exposed as a nice-to-have feature without an essential need.

Privileged Functions:

- addRewards
- addReward
- recoverERC20AndUpdateData
- emergencyRecoverERC20
- setVoter
- setMinter
- setOwner

Issue_38	Governance Issue: Contract owner has full control over reward tokens
Severity	**Governance**
Description	Currently, governance of this contract has several privileges for invoking certain functions that can drastically alter the contracts behavior. As an example, governance can simply withdraw all reward tokens via the emergencyRecoverERC20 function.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.
Comments / Resolution	Acknowledged.

Issue_39	Lack of compatibility with transfer-tax tokens will break reward accounting
Severity	**High**
Description	Currently, rewards can be allocated via the notifyRewardAmount function. This function is widely known from the Synthetix Staking Rewards implementation and was modified in such a manner to allocate the deposited rewards for the next upcoming epoch. This functionality however exposes a problem: It does not account for transfer tokens. Therefore, the contract will attempt to distribute more tokens than initially received, rendering the last claim unsuccessful.
Recommendations	Consider incorporating the before-after scheme.
Comments / Resolution	Acknowledged.

Issue_40	Contract design is vulnerable to whale tricks
Severity	**Medium**
Description	Currently, if deposit and reward allocations are made in epoch 1, these are immutable once the epoch has surpassed. Such a design is vulnerable to the following PoC: 1) Alice is a VESwapX whale and has a majority of the VP, she wants Gauge B to receive a majority of the votes but is also a reward hunter. 2) Gauge A and Gauge B both have a relative amount of reward tokens. 3) Alice votes for Gauge A, taking a majority of the pool size. This behavior will disincentivize other users to vote for Gauge A because they will realize that Alice gets a majority of the rewards, hence Gauge A will not receive many votes 4) In the last block before the epoch is incremented, Alice votes with a part of her allocated VP for Gauge B This behavior results in Alice receiving a majority of bribes for Gauge A due to the fact that users are disincentivized to vote for the said Gauge and additionally Alice will receive a normal share of rewards for Gauge B. *This issue can also be transmitted to the VoterV3 contract.
Recommendations	An isolated fix for this issue would be to not allow users to change their votes a specific time frame before an epoch has surpassed (12 hours as example). However, due to the fact that this codebase is widely used and battle-tested, it is important to weigh the issue severity vs security benefits of a battle-tested codebase. Therefore we come to the conclusion that this issue can be safely acknowledged as it is just part of the design choice.

Comments / Resolution	Acknowledged.

Issue_41	Insufficient precision can result in down-rounding and loss of rewards
Severity	**Low**
Description	Currently the contract uses 18 precision for the reward calculation: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/Bribes.sol#L218 if the rewardToken has 6 decimals but the staking token 18 (or even more), this can quickly result in rewards rounding to zero. Example: rewardsPerEpoch[epoch] = 100e6 precision = 1e18 _totalSupply[epoch] = 1_000_000e18 90e6 * 1e18 / 100_000_000e18 = 0.9 Therefore, no rewards will be distributed for this epoch.
Recommendations	Consider increasing the precision to 1e24 (rewardPerToken & _earned).
Comments / Resolution	Resolved.

Issue_42	totalSupply function incorrectly fetches past supply
Severity	**Low**
Description	The totalSupply function is meant to fetch the current supply, similar to the balanceOf and balanceOfOwner functions. This should be done by using getNextEpochStart as the timestamp since the current supply is increased for the next epoch upon deposits. However, it wrongly uses the currentEpochStart as timestamp, thus resulting in an incorrect return value.
Recommendations	Consider being consistent and invoking getNextEpochStart for the timestamp caching.
Comments / Resolution	Resolved.

MonolithicVoter

The MonolithicVoter contract serves as an interface to interact with the VoterV3 and VotingEscrow for specific tokenIds. This contract essentially hosts the privileges for all project partners, ensuring that no sudden disruptions can happen due to the fact that tokenIds are custodied.

To employ this logic the contract must be the owner of the specific tokenId and the operator can assign a specific designated address to each tokenId.
This address can then vote for whitelisted pools and will receive rewards. It is furthermore also possible for the operator to remove the assignment or transfer the tokenId completely out of the contract (revoke).

There are two whitelist mechanisms:

a) Global Whitelist: Any pool address on this whitelist can be used as VP recipient for any vote.
b) Partner Whitelist: Each tokenId has a unique assigned whitelist which allows the voting for pools on this whitelist by the corresponding tokenId.

Both whitelists are inclusive which means a tokenId can vote for a whitelisted pool on any of these lists. These are solely settable by the contract operator.

The following functions are permissionless callable once the tokenId has been transferred to the MonolithicVoter:

a) claimVoterRewards: This function simply claims any outstanding bribe rewards to the designated address for each tokenId
b) poke: This function allows to repeat the previous round's vote configuration and votes for the current round
c) claimRebases: This function simply claims any outstanding rewards from the RewardsDistributor contract for each tokenId
d) extend: This function allows to extend a tokenIDs lock duration to the maximum lock duration of two years. Once per round.
e) maintenance: Aggregates claimRebases, extend and poke into one call

Privileged Functions:

- assign
- setName
- setOperator
- setMultisig
- setWhitelisted
- removeWhitelisted
- setWhitelistForPartner
- removeWhitelistForPartner
- revoke
- elevatedClaimVoterRewards
- execute

Issue_43	Poke function can be griefed to lower VP
Severity	**Medium**
Description	The poke function allows anyone to repeat the latest vote configuration. This can be abused to decrease the overall VP. If we take a look at the _balanceOfNFT function, we realize that the VP not only decreases with each epoch but also inside each epoch. It basically decreases with each second that has passed: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/VotingEscrow.sol#L924 This fact can be abused by malicious actors to invoke the poke function at the very end of an epoch, thus decreasing the VP compared to the initial poke at the beginning of the epoch.
Recommendations	Consider only allowing to poke once per epoch.
Comments / Resolution	Resolved.

Issue_44	Votes can still be executed after designated address was removed
Severity	**Medium**
Description	A previously attached designated address can be trivially removed via the assign function by assigning address(0) as new designated address. This behavior however does not prevent the poke function, as this function can still be called permissionless, voting for the latest configured pools, which is likely not desired once the designated address has been removed. Additionally, it needs to be mentioned that the poke function can be invoked as soon as this contract is the owner of a tokenId and a vote configuration has been set (transfers can only happen during an abstained state).
Recommendations	Consider ensuring that __ownerOf[_tokenID] is not address(0) during the poke function.
Comments / Resolution	Resolved.

Issue_45	Rebase rewards will not be delegated to designated address if tokenId has expired
Severity	**Medium**
Description	The claimRebases function allows anyone to claim tokens on behalf of any tokenId from the RewardsDistributor contract via the claim_many function. Most of the time (or almost always), these rewards will just be allocated to the tokenId if it's still locked. This is to 99% guaranteed due to the maintenance logic. However, in the rare situation where a tokenId is not locked anymore, these rewards will just be transferred to the owner, which is the MonolithicVoter in that scenario: https://github.com/SwapX-Exchange/contracts-rb/blob/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts/RewardsDistributor.sol#L315 Afterwards, a malicious user (designated address for a different tokenId) can exploit this scenario by subsequently calling the claimVoterRewards function with the received token as parameter (as long as the bribe supports it), which will then transfer the stuck token to the designated address, which is in that scenario the malicious actor instead of the designated address of the initial tokenId. *Notably, the claim_many function can also be directly and permissionless invoked on the RewardsDistributor contract to achieve the same result.
Recommendations	Consider executing a loop over all tokenIds, checking for each tokenId that it has not expired. This will however not prevent the direct interaction. Therefore we overall simply recommend ensuring that all custodied tokenIds are permanently max-locked.

Comments / Resolution	Acknowledged.

Issue_46	Lock-out possibility due to lack of address(0) check
Severity	**Low**
Description	Currently, it is possible to set both the operator and the multisig to address(0). In such a scenario, all tokenIds would be permanently locked in the contract.
Recommendations	Consider validating the parameters accordingly.
Comments / Resolution	Resolved.

Issue_47	Lack of safeTransfer usage
Severity	**Low**
Description	The contract uses the standard transfer pattern for ERC20 transfers. This will malfunction for tokens that return false on transfer that do not return a boolean on the transfer.
Recommendations	Consider using safeTransfer.
Comments / Resolution	Resolved.

Issue_48	Lack of ERC721 Interface
Severity	**Informational**
Description	Currently, the contract lacks the interface which is needed to receive NFTs via safeTransferFrom. This can slightly disrupt the UX.
Recommendations	Consider implementing this interface.
Comments / Resolution	Resolved.