BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

SwapX
Staking Airdrop Vesting Token

June 2024

BAIL security

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 - Staking/Airdrop/Vesting/Token
Website	swapx.fi
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/SwapX-Exchange/contracts-rb/tree/c2e6cc77adcb70cf839e7158e4bd10731416b4f9/contracts
Resolution 1	https://github.com/SwapX-Exchange/contracts-rb/tree/d8c647243d7d14683e32a06b11e2f78912b0abff/contracts
Resolution 2	https://github.com/SwapX-Exchange/contracts-rb/blob/d2009e66dbbe605523999efb677d0f9664d1758b/contracts/MasterChef.sol

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	6	4		2
Medium	4	3		1
Low	12	7		5
Informational	4	1		3
Governance				
Total	26	15		11

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

MerkleTreeAndAirdrop

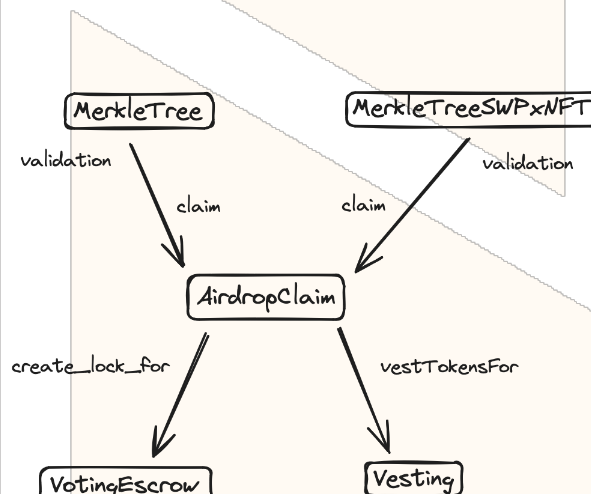
AirdropClaim

The AirdropClaim contract is a custom distribution contract that exposes a claim function which is callable by the MerkleTree and MerkleTreeSWPxNFT contracts, indicating that the validation is completely abstracted to these contracts.
The core functionality of this contract is to distribute the SWPx token via two different pathways:

a) Distribution via Lock Creation: This pathway distributes a percentage of the total amount to a user via simply creating a lock with the maximum lock duration on the VotingEscrow contract.
b) Distribution via Vesting Creation: This pathway distributes a percentage of the total amount to a user via creating a vesting position on the Vesting contract.

The distribution between creating a lock and creating a vesting position is handled by the veShare variable. A veShare of 10_000 indicates 100% and therefore uses the full amount to create a lock, vice-versa a share of 0 indicates 0% and uses the full amount to create a vesting position.

Notably, this contract is meant to be deployed with a corresponding MerkleTree/MerkleTreeSWPxNFT contract.



Privileged Functions

- transferOwnership
- renounceOwnership
- withdrawTokens
- setVeShare
- setMerkleTreeContract

Issue_01	Lack of SafeERC20 usage
Severity	**Informational**
Description	The contract uses the standard ERC20 pattern for transfers/approvals. This will malfunction for tokens that do not follow the IERC20 interface, for example those that return false or do not return a boolean on the transfer/approve call.
Recommendations	This issue can be acknowledged as long as only the SWPx token is used.
Comments / Resolution	Acknowledged.

MerkleTree

The MerkleTree contract is a simple validation contract that allows users to claim tokens from the corresponding airdrop contract (AirdropClaim) with a valid leaf.

It simply validates a user's allocation in the specified merkle tree and then invokes the claim function on the airdrop contract. Therefore, the only purpose of the contract is to validate if a provided leaf is part of a merkle tree, leveraging OpenZeppelin's MerkleProof library.

Every user with a valid leaf can claim once and is then marked as claimed. Furthermore, the contract owner can mark a user as claimed, which is useful for preventing owners with a valid leaf from claiming without the need of changing the whole merkle tree.

Privileged Functions

- transferOwnership
- renounceOwnership
- setMerkleRoot
- setUserClaimed

No issues found.

MerkleTreeSWPxNFT

The MerkleTreeSWPxNFT contract is similar to the MerkleTree contract with the only difference that the allocation is based on a tokenId and only the original minter of this tokenId can trigger the claim process.

Privileged Functions

- transferOwnership
- renounceOwnership
- setMerkleRoot

No issues found.

Tokens

SWPx

The SWPx contract is a simple token contract which is heavily inspired by THENA's implementation. It allows for an initial mint of 40 million tokens, which is triggered by the Minter contract.
Furthermore it exposes a mint function which is triggered by the Minter during every epoch update and mints the desired amount of tokens to the Minter contract, which then further distributes it to Gauges and other contracts.

Privileged Functions

- setMinter
- initialMint
- mint

No issues found.

SWPxNFT

The SWPxNFT contract is a modified ERC721Enumerable contract, which allows users with a valid allocation in the corresponding merkle tree to mint tokenIds. Each mint costs native tokens and must be provided as msg.value with the transaction.

During the contract deployment, the price, max supply and sale timestamp is set. None of these parameters, besides the price, can be changed afterwards. Additionally, the contract incorporates a reservedAmount functionality, which mints NFTs without pushing recipients into the originalMinters mapping, thus not granting them privileges a) & b) below.

Whenever users mint tokens, the minted tokenId will be pushed into the originalMinters mapping, which then grants the original minters several privileges such as:

a) Royalty fee for each epoch
b) Allocation in the MerkleTreeSWPxNFT contract for each minted tokenId

Each address can only mint once.

Furthermore, NFTs can be used to receive rewards from staking within the MasterChef contract

Privileged Functions

- transferOwnership
- renounceOwnership
- withdraw
- setRoot
- setNftPrice
- reserveNFTs
- setBaseURI

BAIL security

Issue_05	Flaw within reserveNFTs will disrupt royalty calculation
Severity	**Medium**
Description	If the reserveNFTs function is invoked while the totalSupply already reached the MAX_SUPPLY, this means no further tokenIds are minted but the reservedAmount variable is increased. Therefore, the totalSupply stays consistent but the reservedAmount increases. This will have the impact that the claimable function within the Royalties contract will be disrupted because the divisor is larger than it should be. (totalSupply does not incorporate the minted reservedAmount) In the worst-case scenario, reservedAmount can even become larger than the current supply, resulting in an underflow revert within Royalties.claimable. **Illustrated:** a) The owner makes multiple calls to reservedNFT incrementing the value for reservedAmount to the point where reservedAmount > total supply. b) User attempts to claim royalties via the claim function. c) The claimable function attempts to calculate the reward for an epoch, with denominator operation as such (_tot - _resAmnt), where _tot is total supply, and _resAmnt is the reserved amount. d) Since step a. caused the reserved amount to become larger than total supply, the calculation for the reward reverts indefinitely. **Illustrated 2:** a) Users interact with the contract as expected, the MAX_SUPPLY is reached.

	totalSupply = 10_000, 1000 NFTs are minted which all can be used to claim royalties. b) Owner invokes the reservedNFT function with amount = 150. No NFTs are minted but reservedNFT is set to 150 c) During the calculation within the Royalties contract, the 1000 tokenIds can claim fees but the divisor is set to 850, effectively making the contract insolvent after 850 tokenIds have claimed their royalties.
Recommendations	Consider reverting in the case where the "_amount" would exceed MAX_SUPPLY, ensuring that no inconsistencies can occur.
Comments / Resolution	Resolved.

Issue_06	Hardcoded multisig address is suboptimal for several reasons
Severity	**Low**
Description	Whenever the owner invokes the withdraw function, the whole native token balance is transferred to the multiSig contract. This address is never changeable which will result in stuck funds for certain scenarios such as lost keys or compromised multisig participants.
Recommendations	Consider transferring the native balance to a _to address.
Comments / Resolution	Acknowledged.

Issue_07	Reentrancy call into reserveNFTs allows owner to bypass MAX_RESERVE
Severity	**Low**
Description	The reserveNFTs function allows the contract owner to mint a specific amount of NFTs to an address. The only difference of this approach vs the standard mint approach is that these mints do not count in for the royalty distribution, hence the totalSupply is also reduced by the reservedAmount to determine the royalty distribution: https://github.com/SwapX-Exchange/contracts-rb/blob/2900185aa9ec23fa1220b92f555344b5881c19c3/contracts/Royalties.sol#L116 The contract exposes an upper limit of 150 tokens to be minted as reservedAmount and this check is enforced at the beginning of the function. However, due to the fact that the CEI pattern is violated and the _safeMint call invokes a hook on the recipient contract, it is possible for a malicious owner to reenter into the same function and bypass the check.
Recommendations	Consider adjusting the reservedAmount before the _safeMint call.
Comments / Resolution	Resolved.

Royalties

The Royalties contract is an epoch-based distribution contract that allows initial NFT minters to receive rewards for each epoch.

The contract incorporates a depositors mapping which allows the contract owner to add and remove addresses to it. Once an address is added to this mapping, it can invoke the deposit function. This function is meant to be called once per epoch and allocates fees to an epoch while simultaneously storing the accurate NFT supply for this epoch, which is then later used to determine the claimable pro-rata amount.

Once the epoch is incremented, which is the case after the deposit function has been invoked, original NFT minters can claim their pro-rata share from the allocated epoch fees, based on the previously cached totalSupply of this epoch and their minted NFT supply.

It is important to mention that it lies solely in the responsibility of the depositors to correctly distribute tokens during the correct epoch times. Since this contract is not connected to the MinterUpgradeable contract, epochs are not validated. Therefore it is possible that multiple epochs can be rewarded at the exact same timestamp.

Privileged Functions

- transferOwnership
- renounceOwnership
- setDepositor
- removeDepositor
- setOwner
- withdrawERC20

Issue_08	Claimable function is completely flawed
Severity	**High**
Description	The rationale of the claimable function is to calculate a pro-rata share from the fee amount for each specific epoch. This is done in such a manner to fetch how much NFTs a user has and calculate for how much % of the totalSupply this accounts for. Afterwards the fee is allocated to the user based on the result of this. Illustrated, this means if a user has 10 tokenIds, the fee is 100e18 and the totalSupply is 100, the user in question will receive 10e18 tokens from the fee, as he owns 10% of the totalSupply. The calculation in itself is correct, however, the problem is that the following call which determines how much tokenIds a user owns: *uint256 weight = swpxNFT.originalMinters(user);* is not epoch specified. Therefore, if there totalSupply is 10 during epoch 0 and a user has no tokenId minted at this point but only at a later point (ie. the user has minted 5 tokens during epoch 2), the calculation in claimable for epoch 0 will still use 2 as weight, while in fact the user has no tokenIds during epoch. **Illustrated:** e) Deposit is called regularly, from epoch 0 till epoch 3 which caches values for feesPerEpoch, totalSupply and reservedAmount for each epoch. f) Alice mints 10 tokens in epoch 3. g) Alice makes a call to claim function which in turn triggers call to claimable to determine the claimable amount. h) Alice has no checkpoints stored yet, so the userCheckpoint mapping will return 0. so the loop starts from epoch 0 to current epoch 3.

	i) Even though Alice only minted tokens in epoch three, the loop uses her NFT weight added in epoch three via originalMinters. j) Alice receives rewards from epoch 0 till current epoch 3. This is a fundamental flaw in the calculation approach and will render the function completely unusable.
Recommendations	A fix for this issue would be to specifically cache which user has minted how many tokenIds during each specific round. The implementation of this fix is non-trivial and an additional nominal fee will apply to validate this architectural change as this does not fall under our standard resolution round. (general refactoring) Another, more simple solution, would be to only allow the activation of the Royalties contract (execution of deposit), once in fact all tokenIds have been minted. This way it is ensured that the totalSupply remains unchanged and the calculation works.
Comments / Resolution	**Failed resolution,** a change in the NFT minting process has been implemented which prevents minting of nfts after the 12th epoch. https://github.com/SwapX-Exchange/contracts-rb/commit/ff0cf183e41a188c2c7ec69872a8285f96cb44db First of all, it needs to be mentioned that this change does not correctly reflect the epoch time, as epochs are always rounded down to the latest point in weeks. Furthermore, this change alone is pointless as it does not prevent the actual described issue. users can still mint in epoch 3 and gain rewards from epoch 0,1,2 (in these epochs the minted tokenId from epoch 3 is not incorporated, thus the accounting is incorrect). A fix needs to prevent calling claimTill before all tokens have been minted. It must be ensured that users can only claim after the minting has been finished completely.

	Summarized, the following steps must be taken: a) Ensure that the epoch start rounds down to the nearest week time (non-critical) b) Ensure that the claimTill function can only be invoked *after* all NFTs have been minted (critical) **Resolution 2: Acknowledged.** The client indicated that royalties will only be deposited *after* all NFTs have been minted. This ensures that users cannot claim earlier.

Issue_09	Claim call can run out of gas if epochs are very far progressed
Severity	**Medium**
Description	As already mentioned within the contract description, whenever the deposit function is called, epochs are increased. There is no safeguard that each epoch lasts 1 week whatsoever. Therefore, it is very much possible for the epochs parameter to grow very large while simultaneously not all users claim each epoch. In such a scenario where a user has never claimed and the epoch is very large, this can result in the claimable function to run out of gas, permanently denying a user to claim his royalties.
Recommendations	Consider incorporating a parameter which allows users to determine up to which epoch the claim should happen. This parameter should be validated to be larger than the last claimed epoch and smaller/equal the current epoch. Furthermore it should be used as a setter for the userCheckpoint mapping.
Comments / Resolution	Resolved.

Issue_10	Deposit before totalSupply > 0 will result in division by zero error
Severity	**Medium**
Description	Currently, there is no limitation when a deposit can happen. In such a scenario where a deposit happens with totalSupply = 0, it would essentially break the contract in later operations because of a division by zero revert.
Recommendations	Consider ensuring that the totalSupply - reservedAmounts is non-zero.
Comments / Resolution	Acknowledged.

Issue_11	Direct transfer of native token will result in locked funds
Severity	**Low**
Description	The contract exposes a fallback receive function which allows for receiving native tokens. These tokens will however remain locked within the contract until withdrawn by governance.
Recommendations	Consider removing the fallback receive function.
Comments / Resolution	Resolved.

MasterChef

MasterChef

The MasterChef contract is a custom NFT staking contract which is heavily inspired by Sushiswap's MasterChef and incorporates a similar reward algorithm. Users can stake their NFTs and receive the SWPx token as reward. Contrary to other NFT staking contracts, all tokenIds have the same weight, thus making the reward distribution fair.

The reward algorithm is slightly customized and allows for two different reward approaches which both distribute the same token:

a) Standard Reward: This is the standard form of reward distribution and is allocated via the setDistributionRate function. The source for these rewards is through the SWPxNFTFeeConverter contract, whenever there are swap fees claimed and distributed, or there are sale proceeds distributed by the NFTSalesSplitter. Rewards which are accrued via this form can be simply harvested directly to a claimer's wallet.

b) Extra Reward: This is an additional form of reward distribution and is allocated via the setDistributionRateExtra function. The source for these rewards solely stems from the Minter contract in the first 12 weeks in the form of team emissions.

Additionally, the contract exposes view-only functionality which keeps track of how many rewards are distributed on each epoch.

Appendix: RewardDebt Modification:

The rewardDebt parameter keeps track of how much rewards would have been allocated to a user based on his amount and the most recent accumulator. This variable exists to ensure users will always only receive rewards based on their last update and the increased accumulator, it is furthermore used to be subtracted from the accumulated reward.

Usually, whenever a deposit or a withdrawal is happening, the accumulator is updated, rewards are claimed based on the updated accumulator and the updated rewardDebt is set.

This contract however incorporates a different methodology upon deposits and withdrawals. First of all, rewards are not distributed upon these interactions and the rewardDebt is only updated with the newly added/removed amount. The historic rewardDebt is not changed. This mechanism was incorporated to ensure that deposits and withdrawals will not falsify current pending rewards.

Illustrated (Deposit):

- accRewardsPerShare = 100e30
- supply = 10
- rewardPerSecond = 1e18

a) Charles deposits 10 NFTs

-> user.amount = 10
-> user.rewardDebt = (10 * 100e30 / 1e12) = 1000e18
-> supply = 20

b) 100 seconds have passed and Charles deposits again 10 tokens (At this point, Charles should receive 50e18 tokens for his historic stake, since he owns 50% of the supply and 100e18 tokens were distributed during this period)

-> accRewardsPerShare = 100e30 + (100e18 * 1e12 / 20)
-> 105e30
-> user.amount = 20
-> user.rewardDebt = 1000e18 + (10 * 105e30 / 1e12) = 2050e18
-> 2050e18

c) Charles calls harvest
-> accumulatedReward = 20 * 105e30 / 1e12 = 2100e18
-> _pendingReward = 2100e18 - 2050e18 = 50e18

Therefore, it is mathematically proved that the deposit function works flawlessly.

Illustrated:

- accRewardsPerShare = 100e30
- supply = 10
- rewardPerSecond = 1e18

a) Charles deposits 10 tokens:
-> user.amount = 10
-> user.rewardDebt = 1000e18

b) Charles waits 100 seconds and calls withdraw for his 10 tokens:

-> accRewardsPerShare = 100e30 + (100e18 * 1e12/20) = 105e30
-> user.rewardDebt = 1000e18 - (10 * 105e30 / 1e12) = -50e18

c) Charles calls harvest:
-> accumulatedReward = 0
-> _pendingReward = 50e18

Therefore, it is mathematically proven that the withdraw function works flawlessly.

However, as from our experience, such modifications are never encouraged and if there is no reasonable need for having such intrusive modification, we often recommend clients to stick to battle-tested code as this ensures that no oddities such as rounding issues can occur.
Due to this recommendation, the client changed the code in accordance to the standard Masterchef logic where rewards are immediately transferred out upon deposit/withdrawals and the rewardDebt is simply set to:

```
user.amount * accRewardPerShare / precision
```

Furthermore, the contract is only meant to be used with the SWPX token as a reward token.

Privileged Functions

- transferOwnership
- renounceOwnership
- setVestingEscrowShare

- addKeeper
- removeKeeper
- setRewardPerSecond
- setRewardPerSecondExtra

Issue_12	Edge-case within updatePool will result in revert due to underflow
Severity	**High**
Description	The updatePool function is the most fundamental function of the whole contract, it essentially handles the correct update of the accumulator to ensure a fair token distribution among all stakers. The mechanism that facilitates this is heavily inspired by the traditional MasterChef mechanism as it simply divides the rewards which are distributed since the last update to the most recent timestamp by the supply: *uint256 reward = time * rewardPerSecond;* *pool.accRewardPerShare = pool.accRewardPerShare + (reward * ACC_TOKEN_PRECISION) / nftSupply;* Since the contract does not work with continuous rewards but relies on the notification of rewards, the possibility exists that there are currently no rewards to be distributed. This mechanism is inspired by the Synthetix Staking Rewards mechanism and is facilitated via the getRightBorder function, as this determines up to which timestamp rewards are allocated. In the scenario where there are no rewards up to the current block.timestamp, it will simply calculate them until the border, ensuring the contract does not attempt to distribute more rewards than actually allocated: *Math.min(block.timestamp, lastDistributedTime);* *uint256 time = rightBoarder > pool.lastRewardTime ? rightBoarder - pool.lastRewardTime : 0;*

Now since the main functionality of the updatePool function is clear, we need to describe an additional functionality which serves for view-only purposes: The introduction of _epochRewards.

This basically just stores information of how many rewards have been allocated to which epoch.

That functionality introduces a critical error: A potential underflow vulnerability in L 354 can result in a revert of the function call and a permanent DoS:

_epochRewardsExtra[activePeriod] = reward - rewardsToLastActivePeriod;

This scenario can occur if rewardsToLastActivePeriod becomes larger than the actual distributed rewards:

*uint256 rewardsToLastActivePeriod = reward * (activePeriod - pool.lastRewardTime) / time;*

Since the activePeriod is not limited by the rightBoarder, the multiplier can quickly become larger than the divisor, resulting in the result becoming larger than “reward” and thus resulting in a revert due underflow.

Illustrated:

pool.lastRewardTime = 604 800(epoch 1)
_lastActivePeriod = 604 800(epoch 1)
rightBoarder = 1 209 600(epoch 2)
activePeriod = 1 814 000(epoch 3)

a) A user calls the updatePool function, rewards from 604 800 to 1 209 600 are calculated and correctly allocated to

	pool.accRewardPerShare b) The epoch functionality enters the else and then if condition: https://github.com/SwapX-Exchange/contracts-rb/blob/2900185aa9ec23fa1220b92f555344b5881c19c3/contracts/MasterChef.sol#L349 rewardsToLastActivePeriod is now calculated, due to activePeriod being larger than rightBoarder, the result will be larger than reward. Therefore, the following calculation will underflow / revert: *_epochRewards[activePeriod] = reward - rewardsToLastActivePeriod;* The provided PoC is just one of many scenarios where this issue can occur. Another example state where this can occur: pool.lastRewardTime = 1 000 000 _lastActivePeriod = 604 800 rightBoarder = 1 100 000 activePeriod = 1 209 600 The root-cause of this issue is the lack of the border incorporation into the epoch functionality. On top of that, the third condition seems to be unreachable: https://github.com/SwapX-Exchange/contracts-rb/blob/2900185aa9ec23fa1220b92f555344b5881c19c3/contracts/MasterChef.sol#L354 This issue is also present within the corresponding view-only functions.

Recommendations	At BailSec, we often emphasize keeping smart contracts as trivial as possible. Any unnecessary complex logic will increase the exploit risk. Specifically since this logic is only used for view-only purposes, we simply recommend removing it, to fix the aforementioned issue and eventually any other issues which are caused by this logic.
Comments / Resolution	Resolved.

Issue_13	Incorrect approval amount to Vesting contract will break harvestExtra function
Severity	**High**
Description	During the harvestExtra function, rewards are not directly transferred to the caller but rather indirectly distributed via: a) Creating a VE position b) Creating a vesting position for b) the approval is incorrect, which results in a DoS of the harvestExtra function in case the share for the vesting position is larger than the share for the VE position.
Recommendations	Consider approving vestingAmount instead of veShareAmount.
Comments / Resolution	Resolved.

Issue_14	Lack of emergencyWithdraw function
Severity	**Low**
Description	Currently, the contract does not expose an emergencyWithdraw function, which can be useful in case there are unexpected issues during the pool update.
Recommendations	Consider implementing a emergencyWithdraw function that loops over all owned tokenIds, transfers these out and resets the rewardDebt.
Comments / Resolution	**Failed resolution,** the tokenIndices mapping is not cleared during the delete call. We recommend clearing all storage variables separately Resolution 2: Fixed The tokenIndices mapping is now cleared.

Issue_15	Lack of upper limit for setRewardPerSecond/Extra can result in stuck rewards
Severity	**Low**
Description	The setRewardPerSecond/Extra functions allow the contract owner to change the rewardRate. This will only have an impact if there is a valid running epoch, as otherwise it would adjust the rewardRate but if there is no running epoch, this rewardRate is not used and will be just disregarded whenever the setDistributionRate/Extra function is called. Therefore, it simply extends or decreases the lastDistributedTime, depending if the new rate is smaller or larger than the current rate. In such a scenario where the new rewardRate is very large, this could result in a loss of rewards because: notDistributed/_rewardPerSecond rounds to zero.
Recommendations	Consider either implementing appropriate validation, ensuring that the result of the division is non-zero or being very conscious about calling this function and setting up a secure governance structure.
Comments / Resolution	Acknowledged.

BAIL security

Issue_16	Call to setDistributionRate and setDistributionRateExtra without token transfer can result in flawed reward accounting
Severity	**Low**
Description	Whenever one of both aforementioned functions is invoked, the amount will be allocated as reward. However, it is not guaranteed that this amount is in fact provided by the caller.
Recommendations	Consider only using these functions for their distinct purposes, ensuring that a sufficient amount of rewards is transferred in.
Comments / Resolution	Acknowledged.

Issue_17	Incorrect _epochRewards allocation in multiple scenarios will result in flawed settings
Severity	**Low**
Description	As already explained in the issue above, the contract incorporates a _epochRewards[epoch] mapping, which represents how much rewards have been distributed/allocated during each epoch. There are two distinct scenarios for this mechanism: a) The MC is in the current active epoch: Allocate all rewards that are happening during this epoch to the current epoch. Flaw: If the current block.timestamp is already in the next epoch, rewards are still allocated to the past epoch. If the epoch is now updated, the epoch beginning may be smaller than the last used block.timestamp. b) The MC is in a past epoch: Allocate all rewards up to the current

	active epoch to the past epoch. Flaw: If multiple epochs have been passed since the last update, all rewards up to the most recent epoch are allocated to the past epoch. Essentially, the issue is that the period is not always sync'd with the Minter period, which means if the _lastActivePeriod is in the past while there have been many period updates in the minter, all rewards are naturally allocated to the _lastActivePeriod. This issue can also be rather a design choice than a real issue.
Recommendations	**As already explained above, since the only purpose of this logic is for view-only reasons and it is not further used in any other contracts of the architecture, we simply recommend removing this mechanism completely.**
Comments / Resolution	Resolved.

Issue_18	balanceOf accounting can dilute reward allocation
Severity	**Informational**
Description	Whenever the accRewardPerShare/accRewardPerShareExtra variables are increased, the reward for the determined period is divided by the current NFT supply in the contract. This ensures that each tokenId receives its rightful share. If however NFTs are transferred to the contract manually (by accident) this will result in some permanently unretrievable rewards because the divisor is higher than it should be.
Recommendations	Consider incorporating internal accounting which increments/decrements a totalNFTStaked variable upon deposit/withdraw.
Comments / Resolution	Acknowledged.

Issue_19	Reentrancy possibility in case of non-standard ERC20 reward token
Severity	**Informational**
Description	The deposit and withdraw functions are vulnerable to reentrancy attacks due to the reward out transfer before the storage update. This allows for draining all rewards from the contract. However, this issue is only rated as informational because the SWPx token will be a standard ERC20 token.
Recommendations	Consider not using any non-standard ERC20 token or implementing a reentrancy guard.
Comments / Resolution	Acknowledged.

Issue_20	Inconsistent rounding direction within deposit and withdraw can result in negligible manipulation of rewardDebt
Severity	**Informational**
Description	Whenever users deposit NFTs, the rewardDebt is increased as follows: *tokenIds.length * pool.accRewardPerShare) / ACC_TOKEN_PRECISION* Notably, this calculation rounds down. Whenever users withdraw NFTs, the rewardDebt is decreased with the same calculation. Rounding down means nothing else than truncation and this only happens if the division is not evenly. Therefore it is possible to round down during deposit and not round down during withdrawals.
Recommendations	We do not consider this issue as a dramatic one because the rounding gain will be negligible. However, we encourage the development team to execute further tests with this in mind.
Comments / Resolution	Resolved, this practice has been removed.

Conversion

NFTSalesSplitter

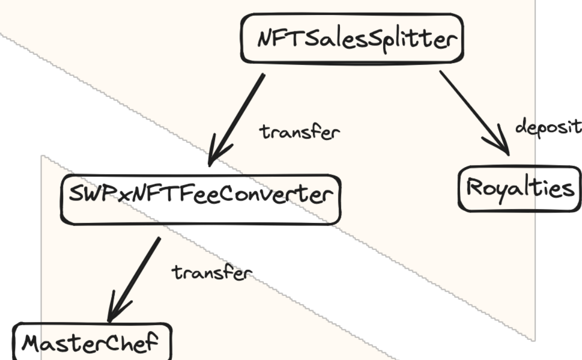
The NFTSalesSplitter contract is a very trivial distribution contract which allows the contract owner to add addresses to the splitter mapping. Once an address is successfully added to that mapping it can invoke the split function which distributes the full contract balance of the native token to:

a) SWPxNFTFeeConverter, where it is then further distributed to the MC as a reward for stakers

b) Royalties, where it is allocated as a reward for original NFT minters

The distribution is dependent on the converterFee and royaltiesFee settings.

As already mentioned within the Royalties contract section, splitters must act responsibly when calling the deposit function on the Royalties contract as this will increment epochs.



BAIL security

Privileged Functions

- transferOwnership
- renounceOwnership
- setConverter
- setSplitter
- setRoyalties
- withdrawERC20
- setFees

Issue_21	Incorrect validation within setFees function can permanently brick the contract
Severity	**Medium**
Description	The setFees function allows for setting the converterFee and royaltiesFee. The following check is employed as safeguard: require(converterFee + royaltiesFee <= PRECISION, 'too many'); This check is incorrect as it only validates the current fees instead of the new fees. In the scenario where the new fees are > 10_000, this check will pass but then has the effect that it can never be changed again.
Recommendations	Consider validating the input parameters.
Comments / Resolution	Resolved.

Issue_22	Split function might run out of gas if pairs/tokens array within SWPxNFTFeeConverter grows too large
Severity	**Low**
Description	Whenever the split function is called, this will invoke the claimFees function on the SWPxNFTFeeConverter contract, which then calls the claimFees function on all pairs in the "pairs" array. The same applies to the swap call. This will not work if the array becomes very large, DoS'ing the function flow.
Recommendations	Consider ensuring that the pairs/tokens array within the SWPxNFTFeeConverter does not grow too large.
Comments / Resolution	Acknowledged.

SWPxNFTFeeConverter

The SWPxNFTFeeConverter is a simple fee handler contract with the main purpose of claiming swap fees from each pair, swapping these fees to the SWPx token and distributing the proceeds to the MasterChef contract. This logic is facilitated by the claimFees and swap functions which are callable by so-called keepers.

This contract is highly configurational by governance such as

a) Adding/removing pairs
b) Adding/removing tokens
c) Configuration of swap paths
d) Adding and removing keepers

Furthermore, besides the keeper logic, governance can manually swap via the Algebra and Solidly router.

Additionally to the fee claim mechanism on V2 pairs, this contract will automatically receive fees from V3 pairs during the claimFees call on the CLFeesVaultV2 contract and a partial amount of the NFT sale proceeds via the NFTSalesSplitter contract, all these fees are swapped to the SWPx token once either keepers or governance becomes active.

Privileged Functions

- transferOwnership
- renounceOwnership
- claimSingleFee
- updateMasterchefDistributionRate
- addPair
- removePair
- addToken
- removeToken
- setPathToToken
- removePathFromToken
- withdrawERC20
- setKeeper

- removeKeeper
- setRouterSolidly
- setRouterAlgebra
- setMasterchef
- setRewardToken
- swapManualInputSingleAlgebra
- swapManualV2
- triggerPause

Issue_23	Swaps will never work for tokens with a transfer-tax
Severity	**High**
Description	The contract currently only invokes swap functions without transfer-tax compatibility. Algebra specifically exposes the exactInputSingleSupportingFeeOnTransferTokens function which invokes the swapWithPaymentInAdvance function on the specific pool. The same applies for Solidly's router with the swapExactTokensForTokensSupportingFeeOnTransferTokens function The idea behind these functions is to support tokens where the input amount can only be determined once it has been received.
Recommendations	Consider using Solidly's swap for transfer-tax tokens as governance function which then allows to manually swap these transfer tokens to a non-transfer-tax token and then swap this token to SWPx via the standard flow of the swap function. Another alternative might be to change the _swap function to just invoke the swapExactTokensForTokensSupportingFeeOnTransferTokens function on the V2 router.
Comments / Resolution	Acknowledged.

Issue_24	Incorrect approval setting within _swapManualV2 will DoS swaps using the Solidly router
Severity	**High**
Description	Within the _swapManualV2 function, the amount is approved to the algebraRouter instead of the solidlyRouter. If the solidlyRouter was changed beforehand, this means there is no approval for existing tokens, rendering the whole V2 swap unusable.
Recommendations	Consider approving the amount to the solidlyRouter instead of to the algebraRouter.
Comments / Resolution	Resolved.

Issue_25	Failed swap will break _swap function
Severity	**Low**
Description	The _swap function was developed in such a way that failed swaps do not disrupt the whole function (due to the try-catch call). However, due to the require statement, such a failed swap will still revert the whole call.
Recommendations	Consider removing this check.
Comments / Resolution	Resolved.

Issue_26	Lack of slippage protection
Severity	**Low**
Description	Within the _swap function the determined minimum output amount is set to zero. This allows MEV bots to sandwich trades and will result in a loss for the protocol.
Recommendations	Due to the fact that this function will be called probably multiple times per day and the fact that keepers are presumably handled via Chainlink automation, thus not being capable of providing such a parameter as function input parameter, this issue can be acknowledged.
Comments / Resolution	Acknowledged.

Issue_27	Router change will not reset approvals
Severity	**Low**
Description	The contract owner can change the Algebra and Solidly router via the setRouterSolidly and setRouterAlgebra functions. These router changes will not reset the existing approvals.
Recommendations	A fix for this issue would be to loop over all existing tokens and revoke their approvals. However, that could introduce another problem if the tokens array grows very large, thus DoS'ing router changes. Therefore we recommend just adding a trivial governance function that allows for manually revoking approvals in such a scenario where a router turns malicious (ie. through proxy update).
Comments / Resolution	Resolved.

Issue_28	addToken function can result in falsified tokens array
Severity	**Low**
Description	The addToken function allows adding new tokens to the tokens array. There is currently no check if the same token is already existing in the array, which would result in duplicate entries.
Recommendations	Consider ensuring that the token is not yet added as a token.
Comments / Resolution	Resolved.

Vesting

The Vesting contract is a trivial vesting contract that allows linear vesting of tokens over a determined time period. Whenever a vesting position is initiated, it will start with linearly vesting tokens once the initial cliff time has been reached.

The contract owner can change the cliff time and vesting period at will and the vestTokensFor function is mainly intended to be used by the MasterChef and the AirdropClaim contract. This contract is only meant to be used with the SWPx token.

Privileged Functions

- transferOwnership
- renounceOwnership
- changeVestingCliff
- changeVestingPeriod

Issue_29	Flaw in claim function allows to drain the contract
Severity	**High**
Description	During the claimable function, it is calculated how much tokens can be claimed from a vesting position. This is done in such a manner to incorporate how much was already claimed and then deduct this from the totalAmount. However, due to the fact that the totalAmount has no upper limit because it is simply depending on the block.timestamp instead of the minimum between block.timestamp and (vestingUnlockStart + vestingPeriodSnapshot), the totalAmount can grow in fact way larger than the maximum vestable amount. An attempt to encounter this is done at the end of the function by ensuring that only the maximum vestable amount can be claimed: *return* *claimableAmount > userVestingInfoMem.amount* *? userVestingInfoMem.amount* *: claimableAmount;* This approach is however insufficient. **Illustrated:** vestingCliff = 0 vestingPeriod = 604 800 With this parameters, any amount will be linearly vested over the course of 604 800 seconds (1 week) a) Charles locks 100e18 tokens at TS 604 800 - UserVestingInfo -> amount = 100e18

	-> start = 604 800 -> claimed = 0 -> vestingCliffSnapshot = 604 800 b) Charles calls claim after 1 814 400 - totalAmount = 200e18 - claimableAmount = 200e18 - return: 100e18 - _userVestingInfo = 100e18 - transfer 100e18 tokens out c) Charles calls claim at the same timestamp once again - totalAmount = 200e18 - claimableAmount = 100e18 - return: 100e18 - _userVestingInfo = 200e18 - transfer 100e18 tokens out Charles therefore successfully stole 100e18 tokens and can further drain the contract if more time passes.
Recommendations	Consider calculating totalAmount with the upper limit as the minimum between block.timestamp and (vestingUnlockStart + vestingPeriodSnapshot). This ensures totalAmount will never become larger than how much the user should actually receive, which is in our example 100e18. Furthermore, it is mandatory to extend testing scenarios for this function.
Comments / Resolution	Resolved.