BAILSEC.IO

OFFICE@BAILSEC.IO

X: @BAILSECURITY

TG: @HELLOATBAILSEC

FINAL REPORT:

Ponzio The Cat

+Audit Round 2

April 2024

Disclaimer:

1. Project Details

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

Project	Ponzio The Cat
Website	Ponzio.io
Type	TOKEN
Language	Solidity
Methods	Manual Analysis
Github repository	https://github.com/PonzioTheCat/contracts/tree/70ffcae9e04666fa65ed8a5f647042afc9a5718f/src
Resolution 1	redacted
Resolution 2	redacted
Resolution 3	https://github.com/PonzioTheCat/contracts/tree/70ffcae9e04666fa65ed8a5f647042afc9a5718f/src

2. Detection Overview

Severity	Found	Resolved	Partially Resolved	Acknowledged (no change made)
High	6	5	1	
Medium	5	5		
Low	9	8		1
Informational	8	5		3
Governance	1			1
Total	29	23	1	5

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

Disclaimer:

Upon the initial audit, our team found a critical vulnerability to drain all tokens in the PonzioVault contract. The team therefore develops a new vault, which will be audited afterwards in a separate audit.

This new audit will not only include the vault itself but also all corresponding interconnections with the rest of the architecture. Additionally, there are quite a few issues which require refactoring of certain functions. If these functions are refactored in such a manner that a casual resolution round is not possible, BailSec reserves the right to charge additionally for the resolution validation.

3. Detection

Token

ERC20Rebasable

The ERC20Rebasable token is a simple rebase token which incorporates a share mechanism for token balances. Trivially explained, the token has a supply of shares which is initially set to 21_000_000e36 and matches with the initialSupply of 21_000_000e18 tokens.

Therefore, initially, 1e18 shares are exactly worth 1 wei of tokens and with time the supply decreases and the amount of shares increases, which means after some time, 1e18 shares will not be worth 1 token anymore. For instance, after the third halving and 12 weeks, 1e18 shares equals 0.098 tokens, which means balanceOf returns zero in such a scenario. Respectively 1.0140e19 shares represent 1 nominal token. This is a design choice but should be kept in mind when dealing with balanceOf operations and small share amounts.

The supply of 21_000_000e18 tokens is however not fixed, as it will decrease over time based on the logic within the Ponzio token contract. Therefore, the ERC20Rebasable contract is designed in such a manner that it can support the decreasing supply, using the shares concept.

Additionally, the total supply of shares will increase as well due to the fee mechanism.

As with all ERC20 tokens, users can transfer tokens via the transfer and transferFrom functions while providing the desired token amount.

For this audit it is important to acknowledge that this token will be not compatible with most DeFi projects. It is only meant to be compatible with UniswapV2 and its own wrapper contract.

Issue_01	Debase mechanism has several side-effects
Severity	**High**
Description	This token is the first of its kind and there is currently no real compatible dex for these tokens. Since it is meant to be used with UniswapV2, there are several issues that will occur: a) Normal swap will revert/result in a loss of funds: During the swapExactTokensForTokens call, the Ponzio token is transferred first to the pair and then the swap function on the pair is invoked. The problem: after the transfer the token is debased and the pair is sync'd, which basically donates the token to the pair. Therefore, the swap will either revert (if minAmountOut is > 0) or the user will have lost their tokens. b) Adding liquidity using the UniswapV2Router will result in lost funds: Upon the addLiquidity function, WETH and Ponzio are transferred to the pair (after the amounts have been calculated): https://github.com/Uniswap/v2-periphery/blob/master/contracts/UniswapV2Router02.sol#L73 Within the _transferShares function however, the updateTotalSupply function is invoked, which then syncs the pair (after the balance of the pair has been increased by the tokens): This will match the reserves with the balance: https://github.com/Uniswap/v2-

	core/blob/master/contracts/UniswapV2Pair.sol#L199 Therefore, both input amounts will be zero: https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol#L114 effectively no liquidity is minted and funds are permanently lost, as the slippage parameters (amountAMin/amountBMin) won't have any impact on this scenario. c) Debase is not happening for the following scenarios: buy Ponzio; burn. During a buy transaction, the corresponding token (USDC/WETH, whatsoever) is transferred to the pair and the swap function is invoked which then transfers out Ponzio. Usually, during the transfer the debase mechanism is invoked which decreases the supply and syncs the pair. This is however not happening due to the lock modifier in the pair. The same issue applies for the burn function where the Ponzio is transferred out. Therefore, users can execute transactions which do not update the supply, exploitable side-effects were not discovered, however, we still recommend further testing for this scenario. d) Non-major liquidity pairs will not be sync'd: The token contract is designed in such a manner to invoke the sync function on the main liquidity pair after a transfer. This will however not happen for any other pairs where the token is used. Contrary to rebase tokens where the balance can be skimmed, this will result in issues if users attempt to use the other liquidity pair for swaps or add liquidity, because the balanceOf is lower than the reserves and users would effectively donate their tokens to the pair during these transactions. e) _mintFee function will become obsolete: The _mintFee function in

	the pair is invoked upon mints and burns and mints ⅙ of the growth of sqrt(k). This means that the growth of k is solely determined in between the periods of adding and removing liquidity, respectively via swap fees or direct donations. The problem is during the debase mechanism, the balance of Ponzio is decreased, this will result in k becoming smaller and therefore no liquidity is minted upon the next mint/burn. This issue is only valid once a debase happens or if the min supply is reached. If swap fees are generated within a period of 4 hours and subsequently mint/burns are invoked, _mintFee is still properly executed. f) Any contract which relies on the invariant: balanceOf = accounting will be broken. This includes 99% of all defi protocols. g) Quote calculation within UniswapV2Library calculates amounts based on non-debased supply, this will have several side-effects.
Recommendations	The main issue hereby is the fact that tokens will be lost upon liquidity addition with UniswapV2 (the token is meant to be compatible with this DEX). Therefore, as already discussed with the development team, a solution would be to not invoke the updateTotalSupply function if the Ponzio contract is transferred to the pair, such that the sync call is avoided. Potential side-effects from this implementation will be elaborated in the follow-up audit.
Comments / Resolution	Partially resolved, the following points do not expose an issue anymore: a), b). Additionally, g) will not expose an issue anymore because of the blacklist mechanism.

Issue_02	Transfers can be executed without supply update
Severity	**Low**
Description	This issue is part of the “Debase mechanism has several side-effects” issue but is slightly more severe, hence we will create a separate issue for it. Currently, there are three entry points on how to transfer tokens: 1) transfer 2) transferFrom 3) transferShares All three entry points finally invoke the _transferShares function, which updates the share balances and eventually triggers the updateTotalSupply function. The try/catch call is vulnerable to gas griefing, which allows a caller to invoke a transfer such that the supply gets not updated (decreased). Such a flexibility should not be granted to users.
Recommendations	Consider incorporating a solution such that the try/catch call cannot be griefed.
Comments / Resolution	Resolved.

Issue_03	Users might transfer less “tokens” than expected
Severity	**Low**
Description	If a user invokes the transfer function with the desired token amount, this will then calculate how much corresponding shares are transferred. If a debase happens before the actual transfer is executed, this will decrease the user’s balance. If now the balance is below the provided amount parameter in the transfer function, the transfer will still transfer all shares of the user and the recipient will receive less “tokens”. On the other hand, if a user has more tokens in his wallet, he might end up transferring the desired token amount but now more shares than expected. This can lead to several side-effects for third party projects and for the user experience.
Recommendations	This is a design choice. Fixes for this issue would be to adjust the _transferShares function to not allow for a larger input parameter than the actual shares and a slippage parameter in the original transfer function. That would however break the ERC20 specification.
Comments / Resolution	Resolved.

Issue_04	Incorrect parameter within transferShares result in falsified event
Severity	**Low**
Description	Within the transferShares function, the internal _transferShares function is invoked which accepts the following parameters: a) from b) to c) shareAmount d) tokenAmount The last parameter is however provided as follows: tokenToShares(shares) which is incorrect, it should be sharesToToken(shares). This will result in an incorrect event output. Fortunately, this is only manipulating an event related parameter, in another situation this can quickly escalate to a critical issue.
Recommendations	Consider using the sharesToToken approach.
Comments / Resolution	Resolved.

Issue_05	Revert protection within _transferShares is pointless for full halving scenario
Severity	**Informational**
Description	Currently, the _transferShares function reverts if the user attempts to transfer more than 2x of his total owned shares. This is to prevent a revert scenario if a user invokes a transfer right before the debase and the full debase happens before the transaction is going through. We will illustrate it with an example: 1) Alice wants to transfer 100 tokens and has 100 shares, she invokes the transfer function 2) In the meantime, the halving period has passed and the supply has decreased, this means now Alice would need to transfer 200 shares to satisfy her 100 token condition. 3) Obviously Alice only owns 100 shares and cannot transfer 200 shares. This is when the protection scenario comes into play: The transfer will still go through but it will only transfer 100 shares, which translates into 50 tokens. The problem in this scenario is that the minted fee was not incorporated which increases the total supply of the shares. In reality, 13.37% of the decreased supply is minted, which then also increases the share amount which is necessary to transfer 100 tokens. Therefore, the _transferShares function would still revert because it is invoked with >200 shares.
Recommendations	The impact of this issue is very limited because the likelihood of such a situation to happen is very low (4 weeks without update). However, for consistency reasons this is still necessary to mention, in our opinion, it is not necessary to alter the codebase.

Comments / Resolution	Resolved, this logic has been removed.

Issue_06	sharesToToken function rounds to the nearest integer
Severity	**Informational**
Description	The sharesToToken function includes an operation to translate an amount of shares into the corresponding token amount. Instead of simply using the pro-rata formula, a different implementation is used. This implementation was incorporated to round to the nearest integer. As example (using example numbers to simplify the issue): totalSupply = 100e18 _totalShares = 1000e36 userShares = 100.000000000000000001e36 This will translate into 10000000000000000000 tokens BUT: userShares = 100.000000000000000006e36 This will translate into 10000000000000000001 tokens Therefore, users can trick the calculation to receive slightly more tokens than they should. On the first look this seems to be negligible. However, many famous protocols have already suffered exploits due to that issue. For instance, the Radiant exploit was due to this issue, because the WadRayMath library also uses the approach to round to the “nearest integer”. Roundings should be checked very carefully and always be against

	the user.
	
	Additionally, the last condition should be removed as well:
	
	`if (tokenAmount_ == 0) {`
	`tokenAmount_ = 1 wei; // user balance cannot fall below 1 wei due to debase`
	`}`
	
	This issue is only rated as informational because it is an event-related parameter.

Recommendations	A fix is not deemed necessary. If however any future implementations rely on the correctness of the event parameter, it would be necessary to round solely down instead of up or down.
Comments / Resolution	Resolved.

Issue_07	Approval can become problematic
Severity	**Informational**
Description	The approve function allows to specify how much tokens can be consumed by a third-party. This can become problematic due to the debase mechanism. If a user for example has 100 shares worth 100 tokens and approves 50 tokens, after the next halving, the spender can consume all 100 shares.
Recommendations	This is likely a design choice but should be kept in mind by users.
Comments / Resolution	Acknowledged.

Ponzio

The Ponzio contract is the debase implementation for the ERC20Rebasable contract and takes care of the balance reduction algorithm. The heart of this contract is the computeSupply function which calculates the debase of the current supply since inception. We will include an appendix for the debase algorithm.

On top of the decreasing supply, there is a fee which is minted as shares to the designated feeRecipient address. There are two scenarios for this fee:

a) totalSupply has not yet reached MINIMUM_TOTAL_SUPPLY:

13.37% of the reduced balance is minted to the feeRecipient.

b) totalSupply has reached the MINIMUM_TOTAL_SUPPLY:

13.37% of the MINIMUM_TOTAL_SUPPLY is minted to the feeRecipient, once per halving epoch fully or partially if an epoch has not yet concluded.

The feeRecipient is meant to be the Stake contract, which is using these tokens as reward tokens for LP token stakers.

Appendix: Debase algorithm

Most algorithms in this format gradually decrease the amount over time using a pro-rata approach. The problem with this approach is that it is imprecise because multiple reductions would not result in the same value as 1 deduction per halving. The reason for this is that the pro-rata depreciation is always calculated on the "up to this time" deprecated value.

The Ponzio team however implemented a more sophisticated approach, which is based on the inception. This ensures that the tracking is efficiently accurate.

Here’s how they did it:

1) The passed time since deployment is calculated
2) The totalSupply since the beginning of the current halving period is determined (based on the INITIAL_SUPPLY and the total concluded halving periods)
3) The totalSupply is decreased pro-rata based on the time which has passed since the start of this epoch.

The clue hereby is that this is always calculated based on the INITIAL_SUPPLY and not based on the current totalSupply.

This will be done until the MINIMUM_SUPPLY of 10e12 tokens has been reached and the decrease will be as follows:

1) 50% per halving epoch
2) 0.2976% per debase epoch

It is important to mention that the supply is only decreased every 4 hours but not in between debase epochs.

Issue_08	Governance Issue: Change of UniswapV2Pair will de-sync main pair and can break the architecture
Severity	**Governance**
Description	The contract exposes a setUniswapV2Pair function which allows to set a new UniswapV2Pair. This could result in sync issues if a new pair is set which is not the main pair. Additionally, if the UniswapV2Pair is set to a contract that does not expose the sync function, the updateTotalSupply function will always revert.
Recommendations	Consider incorporating a Gnosis Multisignature contract as owner and ensuring that the Gnosis participants are trusted entities.

Comments / Resolution	Acknowledged.

Issue_09	Initial vault deposit will result in permanently locked rewards
Severity	**High**
Description	Currently, upon the initialize function, 1000 WEI of LPToken are transferred from the caller and deposited into the Stake contract. The comment indicates that this is to prevent any state of locked rewards if rewards are accrued before the first deposit happens. This is wrong, if a deposit happens and rewards are already accrued, the _getUpdatedRewardsPerShare function will return early and not update accRewardPerShare, hence all rewards which have been accrued until this timestamp will only be allocated in subsequent updates and never lost. This means the first depositor will inherently receive all rewards which have been accrued in hindsight, this is an acceptable fact. However, with the current implementation the first deposit is made by the Ponzio contract (in fact, this is not granted, users can theoretically deposit before). This means any rewards which are accrued until the real first user deposit (not the Token's deposit) will be permanently allocated to this deposit (the Token's deposit). Since the token contract has no harvest nor withdrawal function, these tokens are permanently lost.
Recommendations	Consider not executing the first deposit on behalf of the Ponzio contract.
Comments / Resolution	Resolved.

Issue_10	Function realBalanceOf will underflow if fees are larger than newTotalSupply / returns zero if fees are equal to newTotalSupply
Severity	**Medium**
Description	The realBalanceOf function is specifically designed to not only incorporate the decreasing totalSupply but also incorporate the minted fee on top. This is done in such a manner that the fees are deducted from the totalSupply, which equals the same mechanism as increasing the totalShares by the share amount for the minted fees: shares * (newTotalSupply - fee) / totalShares shares * newTotalSupply / (totalShares + (totalShares * fees / (newTotalSupply-fees))) The problem hereby is the fact that within edge-cases, the fee can become larger than the totalSupply and therefore the function will revert due to an underflow. This special case is considered within the updateTotalSupply function but not the computeSupply function. This issue has been rated as medium severity because it is a view-only function, which can affect only third parties. Additionally, it will return zero if the fees are equal the newTotalSupply
Recommendations	Consider incorporating this edge-case and adjust the calculation to reflect the “if (fees > newTotalSupply) “

	scenario within the updateTotalSupply function. Additionally, as in a previous recommendation, the condition should be extended to "if (fees >= newTotalSupply)". Since this would then just mint the totalShares amount again without incorporating the total share increase, one could then simply follow this pattern during that condition: sharesOf * newTotalSupply / (_totalShares*2) This would then fix the zero return issue. Additional tests should be carefully executed
Comments / Resolution	Resolved.

Issue_11	Function realBalanceOf does not consider scenario where feesCollector is address(0)
Severity	**Medium**
Description	Currently, the realBalanceOf function incorporates a debase and fee minting into the calculation. This is however wrong if the feesCollector is address(0), as this would mean the balance is simply based on the amount of shares and the current supply. Similarly as in the above issue, this is only rated as medium severity because the only impact would be on third-party providers
Recommendations	Consider incorporating this scenario.

Comments / Resolution	Resolved.

Issue_12	Division by zero will result in revert of updateTotalSupply
Severity	**Low**
Description	Currently, the updateTotalSupply function can enter two different conditions to mint fees: Either the fees are larger than the newly updated supply or the fees are smaller. In the latter case, the following calculation is executed: (totalShares() * fees) / (newTotalSupply - fees)) This will result in a zero division revert if fees are exactly equal to newTotalSupply.
Recommendations	Consider changing the condition 1 to: (fees >= newTotalSupply)
Comments / Resolution	Resolved.

Issue_13	Change of feesCollector will prevent debasing in hindsight
Severity	**Low**
Description	If a debase has not happened for some time due to a lack of activity, it is enforced that on the next transfer the total supply is updated. If however the feesCollector is set to address(0) before any interaction triggers an update, this will prevent the rightful debasing in hindsight.
Recommendations	Consider if that is a desired scenario, if not consider invoking the updateTotalSupply function upon the feesCollector change.
Comments / Resolution	Resolved.

Issue_14	Suddenly activated fee address can result in drastic balance altering
Severity	**Low**
Description	The updateTotalSupply function only updates the supply if the feeCollector is non-zero and if the fee is non-zero. In any other scenarios, the supply is not updated and no debase happens. If for example the feeCollector is set to zero for a long time and then eventually set to a real address, this will have two side-effects: a) Decreasing the totalSupply b) Minting the fee Now we must understand that the totalSupply decrease will not make tokens less valuable, if for example the pair has 1_000_000 USDC

	and 1_000_000 TokenIQ and a halving period is passed, all balances will be halved but the price will double. Therefore, there is no nominal loss for users here. The problem arises due to the minted fee which will increase the totalShares. While this will also decrease the balanceOf for users and the pair in the same manner, it has no direct effect on the price but an indirect effect, if these minted shares are dumped into the pair. So users could effectively suffer a disrupting loss in such a scenario.
Recommendations	Consider not allowing the feeAddress to be the zero address to prevent such a scenario.
Comments / Resolution	Resolved.

Issue_15	Loss of fees in certain scenarios
Severity	**Informational**
Description	The contract mints 13.37% of the reduced supply per halving as fees to the Stake contract which then serves as reward incentive for LP stakers. There are two scenarios where less fees are minted than anticipated: a) In case where the fees are > current token supply: This scenario is likely to prevent an underflow within the otherwise happening arithmetic operation. The problem: Less fees will be minted than anticipated. b) If the MINIMUM_SUPPLY is reached and more than 4 weeks have passed, only fees for 4 weeks are minted.

	c) If the totalSupply_ becomes smaller than the MINIMUM_TOTAL_SUPPLY, fees are only calculated based on the difference of previousTotalSupply and totalSupply_, while theoretically, the part which brings the supply below the MINIMUM_TOTAL_SUPPLY should be used to calculate the fee based on the else condition, additionally.
Recommendations	These are likely design choices, therefore we will only mention this as an informational issue and do not require any change. Also it is clear that this algorithm was intensely tested and therefore altering it would require going through the whole process once again.
Comments / Resolution	Acknowledged.

Staking

Stake

The Stake contract is a modified masterchef contract which allows users to stake the IQ/WETH LP pair. Users can interact with the contract as follows:

- deposit: Users can deposit a LP token and receive rewards as incentives
- withdraw: Users can withdraw their deposited LP tokens
- harvest: Users can harvest their accrued rewards
- reinvest: Users can compound their accrued rewards by providing ETH which then atomically creates the LP token and deposits it into the contract

Upon all interactions, the contract's state will be updated which invokes the updateTotalSupply function on the Ponzio contract, decreases the supply, mints and wraps the corresponding fee with the result of the wrapped token being received by the Stake contract. The received amount of wrapped tokens is then allocated amongst all staking positions using the rewardPerToken approach.

The Stake contract therefore forms a closed system with the Ponzio and PonzioVault contract.

Issue_16	Edge-case of pausing can result in permanently locked funds
Severity	**High**
Description	Currently, the Stake contract is developed in such a manner to equally distribute all minted fees to all staked users by using a masterchef-like algorithm. Upon each interaction with the contract, the _harvest function is invoked which will then update the total supply of the IQ token, mint and wrap the corresponding fees to the Stake contract and then allocates the received rewards based on each staked token within the _getUpdatedRewardPerShare function.

	This mechanism looks very straightforward, however, if we incorporate the updateTotalSupply function within the Ponzio contract in this flow, we will quickly realize that this mechanism can be broken. It is important to understand that the debase mechanism of the Ponzio contract can be paused by setting the feesCollector to address(0). Consider the following scenario: 1. feesCollector is the zero address (debase paused) 2. Users interact with the stake contract which then triggers the _harvest and corresponding reward update logic. Due to the feesCollector being zero, the _previousTotalSupply /_ previousUpdateTimestamp variables are not updated and no fees are minted, this is correct and desired thus far. 3. The problem: within the _getUpdatedRewardsPerShare function, the following logic is implemented. This will invoke computeSupply and return the corresponding fees. Usually, the return value will be zero due to the already updated _previousTotalSupply/_previousUpdateTimestap variable: This mechanism is likely for the view-only purpose. However, in our current flow, the returned fees variable is incorporated in the reward allocation. Which means that accRewardPerShare is increased while no tokens actually have been minted to the contract. Withdrawals are therefore blocked until someone manually transfers

	tokens to honor the reward payout or the feesCollector is set back to the Stake address and a real update has happened. Another side-effect is that rewards are actually allocated to users, which means they are capable of harvesting, while they should not, effectively stealing the pending harvests from other users (if there was an existing balance beforehand). If the feesCollector address is however set to another address, for instance to another Stake contract, funds within this contract are permanently locked, because no funds will be transferred in. The root-cause of this issue is the fact that the view-only logic is incorporated in the normal reward flow.
Recommendations	The simplest solution would be to mark the feesCollector immutable. Otherwise it would work to abstract the view-only logic in a separate function and remove it from the normal flow. **We recommend creating two separate functions, one for the update purpose and one for the view-only purpose.**
Comments / Resolution	Resolved, the feesCollector cannot be the address(0) anymore. Additionally the reward mechanism was refactored in such a manner that only the actual received fees will be incorporated as rewards. A separate view-only call-path has been incorporated that is not mingled with the actual reward mechanism anymore.

Issue_17	Returned fees during computeSupply within the _getUpdatedRewardsPerShare function can be incorrect
Severity	**High**
Description	In a previous issue, we have elaborated the impact if the feesCollector address is set to address(0). We have also examined that withdrawals can succeed if the feesCollector is set back to the Stake address. There is an additional unconsidered scenario in case feesCollector is address(0): The computeSupply function will return the fees without upper limit: This is incorrect because in the scenario where the fees are larger than the supply, only the equivalent of the current shares is minted. This means that the accRewardsPerShare is updated using the full fee amount and _lastRewardAmount is set using the full fee amount, while in reality the fee is capped. In such a scenario where the fee is larger than the supply, this will incorrectly increase the accRewardPerShare. Once the feesCollector is then set back to the Stake address, the “correct” fee is minted (the equivalent to the current share supply), which is smaller than the previous allocated amount. This will result in an underflow within the accRewardPerShare calculation, because the currentRewardAmount is now smaller than _lastRewardAmount, effectively breaking the whole architecture. Additionally to that, the previewDeposit function will return an incorrect value because the balance is not yet decreased and therefore the “fee” is converted to the vault’s shares using an unupdated exchange rate.

	The impact is **high** compared to the "View-only pendingRewards function will return incorrect value due to unupdated state" issue, because in that scenario this will not only influence the view-only case. An interesting point about this issue is that users could abuse this flaw to their own advantage. Due to the fact that the accRewardsPerShare is more increased than it should be, users could essentially claim more rewards than they should be able to (if there was a previous balance), effectively rendering other users without claims.
Recommendations	**We recommend creating two separate functions, one for the update purpose and one for the view-only purpose.**
Comments / Resolution	Resolved.

Issue_18	Temporarily locked funds if deposits happen before Ponzio contract is initialized
Severity	**Medium**
Description	The Ponzio contract must be initialized manually during the onlyOwner call to the initialize function. Due to the fact that the initialize function requires a part of the LPPair to be transferred in, it is clear that the LP pair is created before the initialize function is invoked. Users can therefore already buy the token and create their own LP pair. This means, they can deposit into the Stake contract even before the Ponzio contract is initialized (the feesCollector is address(0)). This is a problem and will break withdrawals. Consider the following scenario: 1. Alice is the first depositor and deposits some LP tokens into the contract. Due to the fact that she is the first depositor, the _getUpdatedRewardPerShare function will return early and will not set _accRewardPerShare. 2. Bob now executes the second deposit, the _getUpdatedRewardPerShare function will NOT return early because there is already an existent LpSupply, however, due to the blunder that the view-only logic is included within this function, the accRewardsPerShare variable is already increased. The Ponzio contract is however not yet initialized, which means that feesCollector is inherently address(0). The problem hereby is that the contract does not actually have any rewards, which means if either Alice or Bob would like to withdraw, it

	automatically reverts, because the _harvest function attempts to redeem funds from the vault which are not existent. Depositors are therefore only able to withdraw once the Ponzio contract is in fact initialized, the feesCollector is set to the correct Stake contract and rewards are actually distributed. Additionally, if users externally interact with the vault after the rewards have been allocated but before the feesCollector address is set, there could be a mismatch between the allocated vault shares (which have been used to increase _accRewardPerShare) and in reality received vault shares. This issue is likely already fixed with the new vault implementation.
Recommendations	**We recommend creating two separate functions, one for the update purpose and one for the view-only purpose.**
Comments / Resolution	Resolved.

Issue_19	Lack of emergency withdrawal
Severity	**Medium**
Description	As we have already identified within the first issue, there is a very specific edge-case which can result in 100% locked LP tokens. Nowadays, most masterchefs expose an emergencyWithdraw function, even if sometimes there cannot be an explicit revert scenario elaborated, the likelihood that something bad still happens is given. Specifically for this contract (besides of the mentioned edge-case), there are several interactions with other contracts as well as arithmetic operations. In a scenario where these would cause a revert, all funds would be trapped in the contract and can never be withdrawn.
Recommendations	Consider implementing an emergencyWithdraw function.
Comments / Resolution	Resolved.

Issue_20	Griefing: Reinvest can be DoS'd without economical loss for attacker
Severity	**Low**
Description	The reinvest function harvests a user's rewards and redeems it in the vault for Ponzio, afterwards these tokens are paired with ETH and deposited. Any potential excess is refunded back to the user. Many auditors would now argue that you can manipulate the UniswapV2 pair such that the ratio will not be as desired anymore, resulting in a revert. While this is true, we do not see it as a bug because the malicious user would lose some funds on swap fees and potential backrunners. There is however a legit method to DoS this function, by simply depositing 1 wei on behalf of the caller. Because this would harvest on behalf of the caller, this would then result in a revert of the function because the user that invokes reinvest has no balance to harvest until the next debase has happened.
Recommendations	Consider removing the recipient parameter for the deposit function.
Comments / Resolution	Acknowledged.

Issue_21	View-only pendingRewards function will return incorrect value due to unupdated state
Severity	**Low**
Description	The contract exposes a view-only pendingRewards function which calculates the pending rewards of a user based on the current debase progress of the token. While this will not work if feesCollector is zero (as already mentioned), there is also another blunder which serves as a griefing vulnerability. Upon the _getPendingRewards function, the currentRewards variable is incremented by the return value of the previewDeposit function. This relies on the current exchange ratio of the vault. The clue is that users can manually transfer tokens to the vault to alter the exchange rate. In such a scenario, the pending rewards would be larger than they are in reality (afterwards). Another problem with this function is the following: If the fee amount is larger than the updated supply, only the size of the new supply is minted. This is however not correctly implemented within the _getUpdatedRewardsPerShare function. As this check is simply missing. The result would be a totally incorrect, inflated return value. A third point is the fact that within the view-only scenario, the “fees” are not actually minted as shares. Therefore, the balance in the vault is not decreased (due to the overall share increase) which will then alter the return value of the previewDeposit function: assets * totalSupply / totalAssets

	In the scenario where the fees are actually minted, totalAssets will become smaller, which is however not the case here. Also the totalAssets will not be decreased because the supply is not debased during the view-only function. All these facts will alter the return value. This issue will however naturally be fixed with the new vault implementation. The last point is the fact that fees will still be incorporated if the feesCollector is not the current Stake contract.
Recommendations	A fix for this issue would contain a refactoring of the vault to ensure it cannot be manipulated via asset donation. Additionally, the returned fee value of the computeSupply function should be checked against the new _totalSupply and in the scenario where it is higher, it must be limited to the _totalSupply. It goes without saying that these interconnections must be audited in reference to the newly developed vault contract.
Comments / Resolution	Resolved.

Issue_22	Non-internal accounting of lpSupply can result in diluted rewards
Severity	**Low**
Description	As with all masterchef algorithms, rewards are being allocated per deposited token, hence the variable name: accRewardsPerShare. Back in the days, PancakeSwap implemented exactly the same pattern as is used here - rewards are being allocated based on the current contract balance. As masterchefs evolved, this pattern was deemed as inefficient, as the balanceOf return value could be falsified if tokens would have been accidentally sent to the contract. In that scenario, these tokens would get a reward allocation but in reality, no one owns these, which means rewards are lost.
Recommendations	Consider incorporating an internal tracking of the deposited amount. This needs to be incorporated upon deposit, withdrawal and emergencyWithdraw. It must be thoroughly tested to ensure there it does not implement any DoS vulnerabilities.
Comments / Resolution	Resolved.

Issue_23	Idle IQ tokens can be stolen
Severity	**Informational**
Description	If there are any Ponzio within the contract (that have been sent there by accident), a user can simply invoke the reinvest function with 1 wei as msg.value and would then receive all IQ tokens as refund.
Recommendations	We do not necessarily see this as an issue per-se, it is certainly better than just having the funds locked in the contract. However, for consistency reasons this issue should still be mentioned.
Comments / Resolution	Resolved.

Issue_24	Ownable dependency is unused
Severity	**Informational**
Description	The contract incorporates the Ownable dependency for onlyOwner functions. However, there is no privileged function within this contract, rendering this inheritance useless.
Recommendations	Consider removing the Ownable dependency.
Comments / Resolution	Resolved.

Issue_25	Unused variable(s)
Severity	**Informational**
Description	Variables which are unused will unnecessarily increase the contract size for no reason and will confuse third-party reviewers. L 26: address internal constant WETH_ADDR = 0xC02aaA39b223FE8D0A0e5C4F27eAD9083C756Cc2;
Recommendations	Consider removing unused variable(s).
Comments / Resolution	Resolved.

PonzioVault

The PonzioVault is a simple ERC4626 vault from OpenZeppelin, which serves as Wrapper for the Ponzio contract.

Issue_26	Vault is fully exploitable
Severity	**High**
Description	The vault is a simple ERC4626 vault. The calculation of the vault will use the provided amount parameter while the transfer allows to send less. —————————————————————————————— Consider the following scenario: Status Quo: - totalSupply = 21 000 000e18 - totalShares = 21 000 000e36 - Alice owns 2 100 000e36 shares (10% of the supply) - Within the vault there are 2 100 000e36 shares(10% of the supply) - The vault has a share supply of 2 100 000e18 (corresponding to the deposited token amount for 2 100 000e36 shares) - 2419200 have been passed since the last update, which equals one halving period *** THIS ISSUE IS WITHOUT THE INCLUSION OF THE MINTED SHARES TO SIMPLIFY THE PoC, IF INCLUDED; THE CALL COULD BE WITH SLIGHTLY LOWER AMOUNT AND IT WOULD STILL WORK

	*** 1. Alice calls the deposit function on the vault with amount = 2 100 000e18 (note how theoretically Alice does not own that amount anymore but only half of it) https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol#L171 2. Alice frontruns her own deposit by a call to updateTotalSupply, this will reduce the token supply to 10 500 00e18 3. previewDeposit is invoked and Alice's amount is converted to shares: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol#L151 https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol#L225 This will yield the following calculation: (totalAssets fetches the balance of the shares, which is NOW returned in an updated state) assets * totalSupply / totalAssets 2 100 00e18 * 2 100 000e18 / 1 050 000e18 -> 4_200_000e18 shares

Alice would therefore receive 4_200_000e18 shares

- AliceShares = 4 200 000e18
- NonAliceShares = 2 1000 000e18
- TotalShares = 6 300 000e18

3. The transfer function is now invoked with 2 100 000e18 tokens:

totalSupply is now updated at this point:

amount * totalShares / totalSupply

2 100 000e18 * 21 000 000e36 / 10 500 000e18

-> 4 200 000e36

Alice would therefore need to transfer 4 200 000e36 shares

Let’s take a look at the _update function:

amount * totalShares / totalSupply

2 100 000e18 * 21 000 000e36 / 10 500 000e18

-> 4 200 000e36 shares would need to be transferred by Alice

this will work because of the following:

Status afterwards:

- AliceShares = 4 200 000e18
- NonAliceShares = 2 1000 000e18
- TotalShares = 6 300 000e18

- SharesInVault = 4 200 000e36 (20% of the supply)

These shares are now worth

4 200 000e36 * 10 500 000e18 / 21 000 000e36
-> 2 100 000 tokens (20% of the supply)

Note how Alice now owns ⅔ of the vault while she only deposited the same amount of shares which were previously existent in the vault.

This means that Alice owns:

⅔ * 2 100 000e18 = 1 400 000e18 tokens

4. Alice redeems 4 200 000e18 shares:

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol#L210

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol#L151

https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol#L225

shares * totalAssets / totalSupply

4 200 000e18 * 2 100 000 000e18/ 6 300 000e18
-> 1 400 000e18

Alice would therefore receive 1 030 000e18 tokens, which needs to

	be translated to shares now: amount * totalShares / totalSupply 1 400 000e18 * 21 000 000e36 / 10 500 000e18 -> Alice now received 2 800 000e36 shares, while she only provided 2 100 000e36 shares —-- Please note that the above scenario was just the first iteration of this exploit, there are way simpler PoC's as well. The root-cause of this issue is that the vault calculates the to received shares based on the provided assets parameter: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/token/ERC20/extensions/ERC4626.sol#L177 while the token's transfer function allows for transferring up to 2x the owned shares. Users can therefore invoke the deposit function with a balance that they in reality do not own.
Recommendations	Consider refactoring the vault.
Comments / Resolution	Resolved.

Router

Router

The Router contract is a simple helper contract which allows adding liquidity and swapping tokens. The main difference compared to the standard UniSwapV2 router is the fact that the Ponzio contract is updated before each interaction, which ensures that the supply and corresponding balanceOf return values are up to date.

Issue_27	calcLiquidityToAdd implicitly assumes address(WETH) > address(Ponzio)
Severity	**High**
Description	The calcLiquidityToAdd function is used to determine the input amounts based on the pair ratio and the minimum desired amounts. The following scenarios are possible Scenario 1: - Calculate corresponding WETH amount based on desired Ponzio amount and the pair ratio - If corresponding WETH amount is smaller or equal the desired WETH amount and larger or equal the minimum amount, the function will return these values Scenario 2: - Calculate corresponding WETH amount based on desired Ponzio amount and the pair ratio - If corresponding WETH amount is larger than desired WETH amount or smaller than minimum WETH amount, calculate corresponding Ponzio amount based on desired WETH amount

	- If calculated Ponzio amount is smaller or equal than desired Ponzio amount and larger or equal to min Ponzio amount, return these values The problem with this function is the fact that it is implicitly assumed that reserveA = Ponzio and reserveB = WETH. This will not work if address(WETH) < address(Ponzio). The calculation would therefore return falsified values, which will result in transferring incorrect token amounts to the pair and therefore loss of funds.
Recommendations	Consider sorting the reserves to the corresponding tokens. This can be trivially done by using the UniswapV2Library contract: https://github.com/Uniswap/v2-periphery/blob/master/contracts/libraries/UniswapV2Library.sol#L29
Comments / Resolution	Resolved. The reserves are now sorted depending on the Ponzio rank in the pair.

Issue_28	Edge-case will result in loss of funds upon liquidity addition
Severity	**Medium**

Description	Consider the following scenario:
	
	Alice owns 100e18 Ponzio and 100e18 WETH
	
	Alice wants to add liquidity to the pair, she provides the following parameters:
	
	amountWETHDesired = 100e18
	amountPonzioDesired = 100e18
	amountETHMin = 50e18
	amountPonzioMin = 50e18
	
	The pair has currently the following balance:
	
	Ponzio = 200_000e18
	WETH = 100_000e18
	
	The updateSupplyAndAddLiquidity function is now invoked and the token is updated
	
	After update in pair:
	
	Ponzio = 100_000e18
	WETH = 100_000e18
	
	After update Alice's balance:
	
	Ponzio = 50e18
	WETH = 100e18

Upon the quote function, this will return the following amounts:

amountPonzio = 100e18
amountWETH = 100e18
It will now transfer:

a) 100e18 WETH to the pair
b) 100e18 Ponzio to the pair

Alice however only has 50e18 Ponzio left (due to the updated supply). The transfer will however not revert but it will transfer 50e18 in. (ignoring the fee now for simplicity reasons)

The pair will now only mint tokens based on the provided 50e18 WETH and 50e18 Ponzio:

https://github.com/Uniswap/v2-core/blob/master/contracts/UniswapV2Pair.sol#L123

The user effectively donated 50e18 WETH to the pair.

The root-cause of this issue stems from two things:

1) The same issue as with the vault interaction: the transfer will not revert if more than the user's balance is transferred out.

2) The fact that UniswapV2 is incompatible with liquidity addition for transfer tax tokens. The router quotes the amounts based on the original input amounts, the pair however will result less of the transfer-tax token. Therefore, users will unknowingly donate the excess of the corresponding token to the pair.

	The problem hereby is that for UniswapV2 and fee-on-transfer tokens, this is an acknowledged risk. For the Ponzio contract however, there may be instances where the supply in fact debases 50%, which then would mean that 50% of the provided WETH is donated. This may expose a larger problem than with most fee-on-transfer tokens where between 1 and 10% is lost.
Recommendations	We recommend that a check is executed to ensure that in fact the correct amount was received by the pair, otherwise the transaction should revert. It is however clear that this could increase the revert frequencies for this function, for that manner one could eventually incorporate the balance check in such a manner that a small slippage (1-5%) is acceptable. This would then not disturb the user experience.
Comments / Resolution	Resolved, the ERC20Rebasable contract has been adjusted to not allow the down-sizing of shares anymore. This will ensure that the transfer will revert if the balance is insufficient.

Issue_29	Idle Ponzio within router can be stolen
Severity	**Informational**
Description	Currently, the swap function allows an “amountIn” parameter. In combination within the Ponzio’s transfer function, this could be used by users to drain any tokens which have been accidentally sent to the router. It would unfold as follows: 1. The router has 50e18 Ponzio idle sitting in it 2. Alice has 50e18 Ponzio but invokes the swap function with amountIn = 100e18 3. Due to the transfer behavior, only 50e18 tokens are transferred into the router but the call will not revert, the router now has 100e18 tokens as balance 4. The router now approves 100e18 tokens and executes the swap function on the UniswapV2Router with an input amount of effectively 100e18 tokens. 5. The user will get the output amount based on 100e18 tokens while only provided 50e18 tokens.
Recommendations	This seems to be an acceptable design risk because the router is not meant to hold any funds.
Comments / Resolution	Acknowledged.

Audit Round 2

Detection Overview

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

This audit round focuses on the changes made to the first commit with the following goals:

a) Ensure all fixes are properly validated and do not introduce side-effects
b) Ensure additionally added logic does not introduce side-effects.

Token

ERC20Rebasable

The audit for this contract focuses on the changes and the additional implemented logic. Fixes for old issues will be handled in the corresponding old section and newly introduced issues will be added here.

The ERC20Rebasable contract was slightly modified and the following changes were incorporated:

a) _blacklistForUpdateSupply: This mapping allows to set addresses which do not trigger a supply update if they are the "from", "to" or "msg.sender" upon a transfer.

b) SHARES_PRECISION_FACTOR: This variable has been decreased from 1e18 to 1e3, which will increase the protocol longevity due to prevented overflows in different arithmetic operations.

c) transferSharesFrom: This function has been added which is the equivalent as transferFrom but denominated with shares instead of tokens.

d) _transferShares: This function has been adjusted to remove the down-sizing feature. Additionally, the supply update is invoked before the balance adjustment, this ensures that sync will not result in stuck funds within the pair and prevents gas-griefing attacks.

e) sharesToToken: This function has been adjusted and now strictly rounds down

f) _approve: The supply update is invoked before the approval is granted. This will prevent gas-griefing.

Issue_01	transferSharesFrom allows for draining tokens from any address
Severity	**High**
Description	The transferSharesFrom function was incorporated to facilitate the share transfer towards the WrappedPonzio contract. There are not really any other use-cases besides a potential user-to-user usage, which does not seem fundamentally necessary. The problem with this logic is the whole balance can be drained whenever a user invokes transferSharesFrom with 0.9e3 shares, which translates into zero tokens and therefore bypasses the approval check. This allows for draining a user's tokens without consent. Due to the fact that: a) The token supply is steadily decreasing b) The share supply is steadily increasing c) The sharesToToken conversion is as follows: -> shares * tokenSupply / shareSupply This issue will become more and more severe with the progression of the supply decrease, as the multiplier will therefore become smaller and smaller and the calculation result will become zero even with a higher amount of shares. This exploit can then be amplified as follows: a) Repeatedly drain balance from the pair via the approval bypass b) Sync the pair to increase the value of Ponzio c) Dump the Ponzio into the pair and drain WETH or USDC or

	whatever token is used as pair token d) Repeat this exploit
Recommendations	Consider marking the transferSharesFrom function as a privileged function such that it is only callable by the WrappedPonzio contract, as well as ensuring that zero-approvals cannot be exploited. It is already an acknowledged fact that there are slight issues with approvals which can result in more shares being transferred than initially anticipated.
Comments / Resolution	**Not resolved**, the check is insufficient and will not work if shares/supply progress: 1e3 * 198750000000000000000000000 / 211554321962276284798251744880 = 0.934 The following example is 3 days after deployment and allows a malicious user to steal 1e3 shares without having a valid approval. **Resolution 2: Resolved.** A simple condition has been implemented which will automatically set tokenAmount to 1 if it rounds down to zero. This is sufficient to prevent transfer initiations **from addresses with zero granted approvals.** Rounding issues can still occur for existing approvals, however, this should not have any severe impact, at all.

Issue_02	Certain operations might revert due to removal of down-sizing within the _transferShares function
Severity	**Low**
Description	Contrary to the previous version, the _transferShares function will not down-size the share value anymore if it exceeds a user's balance. While this is a favorable approach for the overall security, it can result in some reverts: **Example 1:** 1) Alice has 100 USDC and 100 Ponzio and adds these amounts using the custom router, she is the first person that adds liquidity 2) She provides 90 USDC and 90 Ponzio as slippage parameters, which are usually valid parameters 3) The supply is updated which leaves Alice with 100 USDC / 50 Ponzio (amount is actually irrelevant as long as its <100) 4) It calculates the needed liquidity and returns 100 USDC / 100 Ponzio 5) The transfer in will revert because Alice has insufficient shares (she only has shares to cover 50 Ponzio and not 100) **Example 2:** 1) Alice has 100 Ponzio and transfers these to Bob 2) Another transaction is executed before Alice's transfer and this will decrease the supply 3) The transfer will now revert because it attempts to transfer shares

	equivalent to 100 tokens while Alice does not own the necessary amount of shares **Example 3:** 1) Alice owns 100 Ponzio and wants to swap 100 Ponzio to USDC, using the Ponzio's custom router 2) The router updates the supply (decreased _previousTotalSupply) 3) The router invokes transferFrom to transfer 100 Ponzio tokens from Alice, this will revert because Alice does not own the shares for 100 Ponzio tokens.
Recommendations	In our opinion, this is an acceptable down-side for enhanced security. We do not recommend any further change to this logic. It should however be noted that this side-effect can occur.
Comments / Resolution	Acknowledged.

Issue_03	Balance can become zero / rounds down
Severity	**Low**
Description	It lies within the token’s nature that the balanceOf function will return zero or round down even if an address holds a reasonable amount of shares: This is due to the shares to token / balanceOf calculation being as follows: balanceOf = shares * tokenSupply / shareSupply combined with the fact that the tokenSupply is decreasing and the shareSupply is increasing. If an important address that relies on the balanceOf function (such as the UniswapV2Pair) returns a balance of zero, this basically means that multiple calculations will be flawed or revert due to a division by zero error. Usually, rounding down against the favor of the user is always net-positive to the protocol, which is desired. The balanceOf function however, is also used by the pair to determine the allowed output amount. This means that users can gain a slight benefit by swapping PONZIO to WETH whenever the balanceOf rounds down, as this means that the provided PONZIO is worth slightly more than it should be. Arbitrage (swapping back and forth) should however not be profitable due to the enforced swap fee by the pair.
Recommendations	This risk is part of the design choice and due to the fact that this concept relies heavily on the usage of one dominant pair, this issue seems less likely to happen in practice - at least the division by zero

	scenario.
Comments / Resolution	Acknowledged.

Issue_04	Approval consumption within transferSharesFrom may be higher than anticipated
Severity	**Informational**
Description	The transferSharesFrom function is the equivalent of the transferFrom function, besides the fact that the input value is determined as nominal shares. The corresponding approval consumption however, is still using the token amount. In such a scenario where the token is in an un-updated state, a user could invoke this function with 100e21 shares that translate into 100e18 tokens, which are then consumed. If now however the supply is updated within the transfer, only 50e18 tokens might be received, while 100e18 tokens are consumed by the approval.
Recommendations	This issue seems rather a theoretical issue that stems from the design choice and the interconnection between share transfer <-> token approval.
Comments / Resolution	Acknowledged.

Ponzio

The audit for this contract focuses on the changes and the additional implemented logic. Fixes for old issues will be handled in the corresponding old section and newly introduced issues will be added here.

The Ponzio contract was slightly modified and the following changes were incorporated:

a) _maxSharesReached: This variable is triggered upon the fee minting process and will prevent overflow of the _totalShares variable

b) realBalanceOf: This function has been adjusted to cover the case where fees > supply.

c) initialize: The whole deposit execution was removed

d) _feesCollector: This variable can never be set to zero

e) setFeesCollector: This function will now update the supply and mints the fee to the most recent _feeCollector

f) setBlacklistForUpdateSupply: This function allows to set addresses which should not trigger a fee update when transfers happen

Issue_05	realBalanceOf function will return incorrect value if subsequent update ends fee distribution
Severity	**Medium**
Description	The realBalanceOf function is a view-only function which is potentially used by third-party services that are building on top of the Stake contract. The rationale behind this function is to return the real-time value of the current shares, which incorporates the real-time supply and the real-time fee. Previous vulnerabilities were fixed, however, the edge-case where the next subsequent update will stop the reward distribution is not incorporated. This means, in such a scenario where the next update will trigger an overflow and therefore mark _maxSharesReached, no fees are minted. However, the realBalanceOf function will still incorporate the theoretical fees, while in reality they will not be minted.
Recommendations	Consider simply returning: sharesOf * tokenSupply / shareSupply if the returned fees would result in an overflow of the _totalShares variable.
Comments / Resolution	Resolved.

Issue_06	Fees are potentially not minted if token is a long time in an un-updated state
Severity	**Informational**
Description	A new edge-case has been implemented which will stop the fee distribution, if this would result in an overflow of the _totalShares variable. If the token is a long time in an un-updated state and a lot of fees are accrued, it can happen that these fees will reach the overflow threshold. This will result in no fees being minted at all.
Recommendations	An isolated fix for this issue would be to alter the _mintShares function to mint exactly the amount which is needed until the threshold is reached. However, this fix would need to be incorporated into the realBalanceOf and pendingRewards function as well, requiring additional time-allocation for the resolution round. In our opinion, this issue can be safely acknowledged because it is not likely that the token will be in an un-updated state for such a long time to have a negative real-world impact. It can be seen as the same design choice to mint only the nominal amount of the totalSupply if the fee amount increases the totalSupply, since in this scenario, also some fees will be lost/not minted.
Comments / Resolution	Resolved.

Staking

Stake

The audit for this contract focuses on the changes and the additional implemented logic. Fixes for old issues will be handled in the corresponding old section and newly introduced issues will be added here.

The Stake contract was slightly modified and the following changes were incorporated:

a) _lpBalance: This variable was introduced for proper tracking of the staked amount.

b) Ownable inheritance has been removed as it didn’t serve any functionality

c) reinvest: A before-after check has been incorporated to ensure users can only use their own rewards.

d) pendingRewards: This function has been largely modified for bug-fixing purposes.

e) emergencyWithdraw: This function was newly implemented to allow withdrawals in emergency situations without caring about rewards.

f) skim: This function was newly implemented to retrieve any lp tokens which have been accidentally sent to this contract. It is publicly callable.

Issue_07	pendingRewards function will return incorrect value if subsequent update ends fee distribution
Severity	**Medium**
Description	The pendingRewards function is a view-only function which is potentially used by third-party services that are building on top of the Stake contract. Previous vulnerabilities were fixed, however, the edge-case where the next subsequent update will stop the reward distribution is not incorporated. This means, in such a scenario where the next update will trigger an overflow and therefore mark _maxSharesReached, no fees are minted. However, the pendingRewards function will still incorporate the theoretical fees, while in reality they will not be minted.
Recommendations	Consider setting newRewards and newTotalShares = 0 if the fee amount results in _totalShares becoming larger than uint256.
Comments / Resolution	Resolved.

Issue_08	pendingRewards function will return flawed value if feeCollector is not the stake contract
Severity	**Medium**
Description	In the scenario where the feeCollector is not the stake contract, the computeSupply function is not invoked. The rationale behind this practice is to not accidentally allocate rewards which will not be minted towards this contract. So far, this is correct. A problem however arises due to the fact that the newTotalShares and tokenSupply are not updated. In fact, even if the feeCollector is not the stake contract, the supply will still be increased and fees will be minted - just not to the Stake contract but to whatever other contract/address is used as feeCollector. This has the effect that the previewUnwrap function is invoked with the un-updated share supply and token supply, returning a larger pendingRewards return value than expected.
Recommendations	Consider, under all circumstances using the updated token and share supply.
Comments / Resolution	Resolved.

Issue_09	Theoretical overflow vulnerability within _getUpdatedRewardPerShare may DoS functionality
Severity	**Informational**
Description	The _getUpdatedRewardPerShare function is responsible for increasing the accumulator based on the lpBalance and the reward balance delta. The arithmetic operation is as follows: accRewardPerShare_ = _accRewardPerShare + (currentRewardAmount - _lastRewardAmount) * PRECISION_FACTOR / lpBalance; This may result in an overflow if the delta between currentRewardAmount and _lastRewardAmount becomes excessively large. This issue is rather from a theoretical nature, as the share supply would need to become incredibly high for this to occur, plus updates would need to be prevented/not executed for a long time period.
Recommendations	Consider using the same mulDiv approach as within the rest of the codebase.
Comments / Resolution	Resolved.

Issue_10	Custom token allows for exploiting skim to drain the contract
Severity	**Informational**
Description	If a custom token is used as staking token, which hooks to the recipient before the balance change upon a transfer, a malicious user can reenter on the said hook, invoking the skim function and abusing the un-updated delta between _lpBalance and balanceOf, draining the contract. This issue is only rated as informational because it is clear all along that a standard UniswapV2 lp token is used.
Recommendations	Consider adding a nonReentrant modifier to the skim function. For additional security strength consider adding the same to the emergencyWithdraw function.
Comments / Resolution	Resolved.

WrappedPonzio

A full re-audit is conducted for this contract. The WrappedPonzio contract allows users to deposit the Ponzio token and receive a corresponding amount of shares. It is solely based on the provided share value and ignores any supply changes (for the hard calculation).

The WrappedPonzio contract is a simple wrapper contract which allows users to wrap their Ponzio token to a standard ERC20 token with a static balance. As output amount, users will receive exactly their provided input amount of shares deflated by 1e3, which makes the vault bullet-proof against manipulation attacks.
Therefore, as example, if a user deposits 100e21 shares, he will receive exactly 100e18 WrappedPonzio.

Issue_11	previewWrap may result in slightly less tokens than anticipated
Severity	**Informational**
Description	The previewWrap function converts a given amount of PONZIO tokens to the corresponding amount of WPONZIO, depending on the total amount of shares and total supply. This is done in such a fashion that the return value is not only rounded down via mulDiv, but also truncated by 1e3 amount_ = _asset.tokenToShares(assets, newTotalShares, newTotalSupply) / SHARES_PRECISION_FACTOR; Trivially speaking, this means that the return value of WPONZIO is slightly less than anticipated, which is perfectly fine for the wrap logic. A slight problem may arise because this function is externally invoked by the Stake contract during the pendingRewards function. This will have the effect that the pendingRewards function will return slightly

	less rewards than a user currently owns.
Recommendations	We do not see the necessity of a change for this issue. However, it should be kept in mind for protocols that build on top of this architecture that the pendingRewards function may return slightly less rewards than in fact are accrued up to this point. The difference is however so negligible that a flash-theft (for vaults that are built on top of it) is not practicable.
Comments / Resolution	Acknowledged.

Router

Router

Issue_12	First liquidity addition will not work
Severity	**Low**
Description	Contrary to the previous version, the scenario where both reserves are zero was removed, see UniswapV2 implementation: https://github.com/Uniswap/v2-periphery/blob/master/contracts/UniswapV2Router02.sol#L46 This means that the first liquidity addition will never be possible with this contract, which forces this to be done using a "blacklisted" UniswapV2Router.
Recommendations	Consider checking for the reserve amount beforehand, as done in the regular UniswapV2Router as well. Additionally, consider adding the conventional assert check back to the quote logic.
Comments / Resolution	Acknowledged, the team indicated that the first liquidity addition will be done within the deployment script, using the UniswapV2Router.