yAcademy GET Protocol Staking Review

Review Resources:

- Staking documentation

Residents:

- engn33r

Guest Auditor:

- securerodd

Fellows:

- pandadefi
- spalen
- shung
- prady
- PraneshASP

Table of Contents

Review Summary

GET Protocol

GET Protocol provides a ticketing service that uses ERC721 tokens. This review focused on the GET Locked Revenue Distribution token, which will enable staking of the existing GET token to align incentives of GET token holders with GET's objectives and future trajectory.

The contracts of the GET Protocol Locked Revenue Distribution token were reviewed over 8 days, 1 of which was used to create an initial overview of the contract. The code review was performed between December 5 and December 12, 2022. The code was reviewed by 1 resident for a total of 26 review hours (engn33r 26 hours). In addition, one guest auditor and a number of yAcademy Fellows from Fellowship block 4 also reviewed the contracts and contributed over 60 man hours. The repository was under active development during the review, but the review was limited to the latest commit at the start of the review. This was commit ab272ced94d6bc8cc1ded2664408a3fa7ce67128 for the Locked Revenue Distribution Token repository.

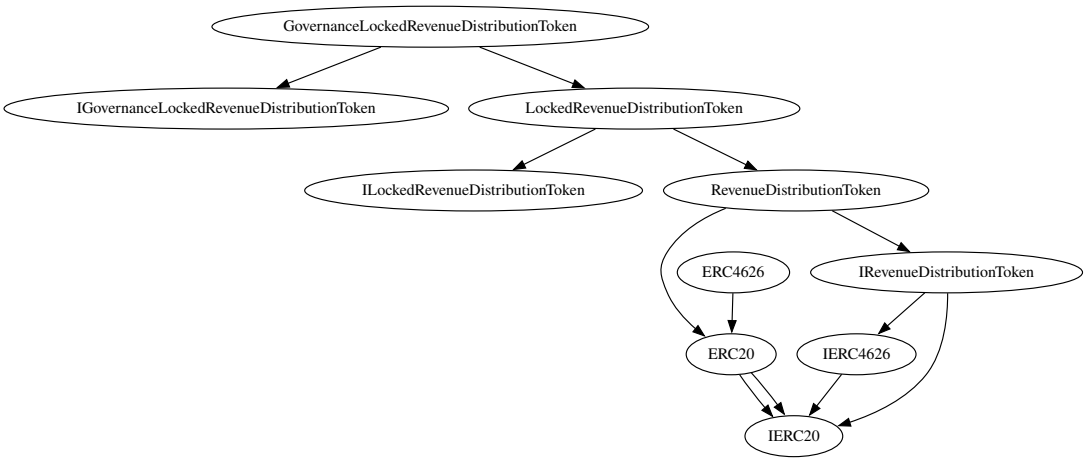
Scope

The scope of the review consisted of the following contracts at the specific commit:

- src/GovernanceLockedRevenueDistributionToken.sol
- src/LockedRevenueDistributionToken.sol
- src/interfaces/IGovernanceLockedRevenueDistributionToken.sol
- src/interfaces/ILockedRevenueDistributionToken.sol
- src/libraries/Math.sol

A key piece to understanding the design of the GET Locked Revenue Distribution Token is understanding the inheritance structure of the contracts,

visualized here. The RevenueDistributionToken contract, borrowed from Maple, provides the ERC20 and ERC4626 structure with a vesting schedule. The contracts inherited by RevenueDistributionToken were also considered in scope. The LockedRevenueDistributionToken contract builds on top of the RevenueDistributionToken contract and adds the withdrawal request and instant withdrawal mechanisms with fees to incentivize stakes to remain staked in the vault. Finally, the GovernanceLockedRevenueDistributionToken contract adds voting and governance controls on top of the other contracts. The vault yield will come from a percentage of ticket fees sold by GET.



Any code not included in these contracts, such as voting and proposal functions, contracts related to yield generation, or the ticket purchasing/selling process, was not in scope for this review. After the findings were presented to the GET Protocol team, fixes were made and included in several PRs.

This review is a code review to identify potential vulnerabilities in the code. The reviewers did not investigate security practices or operational security and assumed that privileged accounts could be trusted. The reviewers did not evaluate the security of the code relative to a standard or specification. The review may not have identified all potential attack vectors or areas of vulnerability.

yAcademy and the residents make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAcademy and the residents do not represent nor imply to third parties that the code has been audited nor that the code is free from defects. By deploying or

using the code, GET Protocol and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Good	Only a few functions are owner-controlled, including `setPendingOwner()`, `updateVestingSchedule()`, `setInstantWithdrawalFee()`, `setLockTime()`, and `setWithdrawalFeeExemption()`.
Mathematics	Good	There is no complex math in the contracts. The rounding up or down in ERC4626 functions is performed as required by the specification.
Complexity	Good	While the layered approach of inheriting other contracts does add complexity to the project, the approach taken on some key functions, such as transferring shares to the vault when a withdrawal request is created, keeps the overall logic clean.
Libraries	Average	Some standard ERC20 libraries as used, while other contracts (GovernanceLockedRevenueDistributionToken.sol, RevenueDistributionToken.sol) are derived from existing contracts borrowed from OpenZeppelin and Maple Finance. Using existing libraries tested on mainnet is good, but adding modifications to these libraries can sometimes introduced unexpected issues.
Decentralization	Good	Besides the few access controlled functions that are controlled by the owner, the protocol is designed to be immutable. The owner cannot steal value or control the shares of the users. No proxies or migration logic is found in the contracts, which can be a positive for decentralization.

Category	Mark	Description
Code stability	Good	The contracts did not have TODOs in comments and looks close to production-ready.
Documentation	Good	The NatSpec comments in GovernanceLockedRevenueDistributionToken and LockedRevenueDistributionToken were thorough and clear. Perhaps one area for improvement is the RevenueDistributionToken contract, which mostly relied on existing comments from Maple Finance but leaves room for improvement.
Monitoring	Good	All important functions and state variable changes emit events with relevant variables for logging purposes.
Testing and verification	Average	The test coverage is over 90% in the core GovernanceLockedRevenueDistributionToken.sol and LockedRevenueDistributionToken.sol contracts. Some branches are not covered by tests, and the tests could introduce additional variety, for example alice and bob not receiving the same number of assets.

Findings Explanation

Findings are broken down into sections by their respective impact:

- Critical, High, Medium, Low impact
  - These are findings that range from attacks that may cause loss of funds, impact control/ownership of the contracts, or cause any unintended consequences/actions that are outside the scope of the requirements
- Gas savings
  - Findings that can improve the gas efficiency of the contracts
- Informational
  - Findings including recommendations and best practices

Critical Findings

None.

High Findings

1. High - Double voting using cross-chain voting (engn33r)

The staking contract is used to reward stakers by taking deposits of GET and providing xGET. The xGET token also provides governance voting privileges. It is documented that there will be a staking contract on Ethereum mainnet and Polygon. While GET is bridgeable between Ethereum mainnet and Polygon, xGET is not bridgeable in order to prevent double voting with the same assets. The problem is that GET is bridgeable, so if a vote is valuable enough for a user, that user can accept the 15% instant withdrawal fee, bridge their GET, restake, and vote again on the other chain.

Technical Details

The process a staker would take to double vote with their assets is:

1. Vote on the first chain (can be Ethereum mainnet or Polygon)
2. Take the 15% instant withdrawal fee
3. Bridge assets
4. Restake GET to receive xGET
5. Vote again on the second chain
6. The end result is 85% more votes than should be possible from the original asset value

There is even more complexity involved in cross-chain voting. Some other cases to consider are:

1. The argument could be made that no user would want to lose 15% of their assets to get extra votes. This would depend on the bribe market and other factors, but there is also a way for the user to recapture some of their “lost” 15%. If a user has X amount of assets staked in the vault, 10% of the total amount could be used for this cross-chain double voting while the other 90% of the assets reclaim some of the “lost” 15% because the 15% fee is distributed among the stakers in the vault. To take this to the next level, a user who knows they will be performing an instant

withdrawal can borrow GET at a certain interest rate, stake it, take the 15% instant withdrawal fee hit, and later withdraw the borrowed GET potentially at a profit before returning it to the borrower. Some modeling or linear programming is necessary to determine exactly when recapturing the 15% fee is profitable, especially in the scenario involving borrowed GET, which will not be examined further here.

2. If a snapshot of user assets at a specific block is used to determine votes held by each address, the question is how to synchronize this point in time across chains. It is unlikely such a process would be exact, given that mining a block is not an instant process. If there is a different in snapshot times between Ethereum mainnet and Polygon, it may enable time for assets to be bridged across chains to enable this double voting. The exact plans around how this would work were not outlined in the smart contract and are not in scope of this review, but should be considered.

Impact

High. The current cross-chain voting design is problematic and no major DeFi protocols have a similar multichain voting system based on a Compound-based voting contract.

Recommendation

It is easier to only allow voting on Ethereum mainnet rather than allowing cross-chain voting. Some protocols with vote escrow tokenomics use the Ethereum-only voting approach, like Curve and Balancer. Compound, the protocol where the governance logic is taken from, is only deployed on Ethereum and therefore does not have cross-chain voting either. One place where the cross-chain voting and GET token security could be improved, at the cost of regular functionality, is changing the GET token on Polygon from a PoS bridged token to a Plasma bridged token. This would add a 7-day withdrawal delay to bridging actions which may or may not be sufficient to remove the ability to perform double voting depending on what checkpoints are used and how long a voting period lasts.

Developer Response

Acknowledged, won’t fix.

There are a number of details with the GET Protocol governance setup that mitigate the risk and impact of this, firstly being that governance does not currently control critical protocol operations on-chain. As a result we have the

opportunity to plan and design around this using our current off-chain governance solution and ensure that this is not exploitable. This will necessitate synchronising snapshots between networks closely enough so that it is not feasible to bridge tokens across networks during that window. Between Polygon and Ethereum it should therefore be feasible to keep snapshots synchronised to a few seconds of eachother. We will continue to design multi-chain governance with this risk in-mind.

2. High - Withdraw and cancel requests can be frontrun for profit (pandadefi)

The contract distributes profit over a week to the users. When a user withdraws, the shares are burned in an instant, and this results in an immediate increase in share price that can be exploited for profit. Here the attacker will profit from the unrealized profit during the duration of the `lockTime`; this is not an attack that is feasible with instant withdrawal. The leftover fee on instant withdrawals is redistributed through the `updateVestingSchedule()` mechanism. The same attack can be performed on cancel request.

Technical Details

In the following test example, Alice deposits funds, profits are scheduled, and Alice makes a withdrawal request while funds are being streamed. Right before the `executeWithdrawalRequest()` is triggered by Alice, Bob deposits funds, and he will be able to get the profits immediately. Because Alice `executeWithdrawalRequest()` is triggered right after the deposit, the leftover profits from the duration of the `lockTime` are distributed to everyone, including Bob, immediately through the burn of shares. In the test, we compare the shares with Eve's shares. Eve deposited right after Alice's withdrawal. They are very different from the amount of Bob shares's. The share amount is very different because the price per share has dramatically increased.

```
function testFrontRunWithdraw() public {
    address eve = address(0x0E5E);

    // first Alice deposit
    _setUpDepositor(alice, 1 ether);
    // We add some profit
    asset.mint(address(vault), 1 ether);
```

```
    vault.updateVestingSchedule();
    vm.warp(block.timestamp + 2 days);
    // Alice starts a withdrawal request
    vault.createWithdrawalRequest(1 ether);
    // we increase time to reach the end of the lock
    vm.warp(block.timestamp + 26 weeks);
    // Bob deposit
    _setUpDepositor(bob, 1 ether);
    vm.stopPrank();
    // Alice execute the withdraw
    vm.startPrank(alice);
    vault.executeWithdrawalRequest(0);
    // Eve deposit
    vm.stopPrank();
    _setUpDepositor(eve, 1 ether);
    vm.stopPrank();

    assertApproxEqRel(vault.balanceOf(eve), vault.balanceOf(bob), 10e6);
}
```

```
function convertToAssets(uint256 shares_) public view virtual override returns
(uint256 assets_) {
    uint256 supply = totalSupply;  // Cache to stack.

    assets_ = supply == 0 ? shares_ : (shares_ * totalAssets()) / supply;
}
```

Impact

High. Monitoring the blockchain for large withdrawals will be beneficial to frontrunners and will reduce the benefits of the fair users, reducing the incentives to deposit GET.

Recommendation

The shares should be burned over a period of time similar to the `updateVestingSchedule()` mechanism.

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#1.

Rather than implementing a burn period for the shares, we instead ensure that tokens are burned at the current conversion rate rather than at differing rates causing an instant distribution. This results in the same outcome with less complexity and resulting rewards remain on the vault contract address for vesting within the next cycle.

Medium Findings

1. Medium - Vault may be susceptible to donation attack (securerodd)

When a user deposits into the vault, the amount of shares they will be issued is calculated by multiplying the deposited assets by a ratio of the existing shares to current assets. The result of this calculation is then rounded down, following the ERC specifications. The problem arises when the value of `totalAssets()` is manipulated to create unfavorable rounding for new users depositing into the vault at the benefit of users who already own shares of the vault. With sufficient capital and a gap in time between the first and second deposit, the first depositor in the vault may be able to leverage a donation attack to prevent future users from receiving any shares for their deposits.

Technical Details

The first user to call `deposit()` on the GovernanceLockedRevenueDistributionToken.sol contract is issued the same quantity of shares as the underlying assets they provide. A user can abuse this to deposit a single quantity of assets in return for one share, setting the `supply` to 1. This same user could then send a large donation of underlying assets to the vault through a direct transfer and call the `updateVestingSchedule()` function to begin increasing the `totalAssets()` value. The ability to call `updateVestingSchedule()` after a donation is a requisite for this attack due to the `totalAssets()` value relying on the `issuanceRate`. The required capital for a successful donation attack will diminish over the course of a 14 day period. The below POC can be added to Compound.t.sol to show the donation amount needed to round another depositor's share issuance to 0 after 1 day has passed between deposits:

```
function testDonationAttack() public {
```

```
    // after set up, allow alice to withdraw all but a dust amount from the vault
    // we warp ahead in time to avoid withdrawal penalties to ensure the vault is left with only dust
    vm.startPrank(alice);
    uint256 allButDust = 1 ether - 1;
    vault.createWithdrawalRequest((allButDust));
    uint256 afterWithdrawalPenalties = start + 26 weeks;
    vm.warp(afterWithdrawalPenalties);

    // execute the withdrawal request
    vault.executeWithdrawalRequest(0);
    vm.stopPrank();

    // print out the shares that would be minted from 1 ether
    console.log("1 ether deposit before donation attack yields _%s_ shares.", vault.previewDeposit(1 ether));

    // provide bob with starting capital
    asset.mint(bob, 14 ether + 1);

    // bob performs the donation. With a 1 day wait, it will require 14 times the capital of the other user to issue 0 shares
    // this is because the vesting schedule follows a linear issuance mechanism over the course of 14 days
    vm.startPrank(bob);
    asset.transfer(address(vault), 14 ether + 1);

    // call the update vesting schedule and wait 1 day for issuance rate to increase
    vault.updateVestingSchedule();
    vm.warp(afterWithdrawalPenalties + (1 days));

    // demonstrate that a user who wishes to deposit 1 ether will now be minted 0 shares
    console.log("1 ether deposit after donation attack yields _%s_ shares.", vault.previewDeposit(1 ether));
}
```

Impact

Medium. The vault could be subject to a DOS attack and some user funds may be at risk of compromise, however the time delay and capital requirement makes this attack less likely to occur.

Recommendation

Incorporate a rounding loss protection mechanism or otherwise ensure that it is economically infeasible for a user to perform a donation attack, such as by guaranteeing a minimum `supply` value during vault deployment. A discussion around this attack and known solutions can be found here.

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#2.

A minimum deposit amount is now required at the time of deployment with the shares burned to `address(0)`.

2. Medium - Redistribute event emits incorrect value and could underflow (securerodd, pandadefi)

The LockedRevenueDistributionToken.sol contract emits the Redistribute event after a withdrawal request is executed. This event is intended to emit the number of assets redistributed to other stakers due to a cancellation or early withdrawal, however, its calculation does not accurately return this value and under certain circumstances it may even revert the transaction.

Technical Details

Within the `executeWithdrawalRequest()` function, a user's withdrawal request is processed, shares are transferred from the vault to the user, and the equivalent number of the user's shares are burned. When this internal `_burn()` function is called, the issuanceRate, `totalAssets()`, and supply values change. The Redistribute event is then emitted and the value it calculates in its `convertToAssets()` call is significantly impacted by these changes. Adding the following POC to InstantWithdrawal.t.sol demonstrates that the skewed value can even prevent the last share(s) from being withdrawn from the vault due to an arithmetic underflow:

```
function testExecuteWithdrawalRedistributeUnderflow() public {
    // Alice deposits 1 ether of underlying asset to the vault
    // Bob deposits 2 ether of underlying asset to the vault
```

```
    _setUpDepositor(alice, 1 ether);
    _setUpDepositor(bob, 2 ether);

    // Allow bob to redeem early as seen in the instantWithdrawalFeeSharing test
    // This leaves positive yield in the vault for alice
    vault.redeem(vault.balanceOf(bob), bob, bob);
    uint256 bobFee_ = _instantWithdrawalFee(2 ether); // == 0.3 ether
    assertEq(asset.balanceOf(bob), 2 ether - bobFee_);
    vm.stopPrank();

    // Attempt to withdraw our balance as alice
    // This will throw an arithmetic underflow due to the incorrect calculation in
    // The Redistribute event
    vm.startPrank(alice);
    vault.createWithdrawalRequest(vault.balanceOf(alice));
    vault.executeWithdrawalRequest(0);
    vm.stopPrank();
}
```

Impact

Medium. Incorrect Redistribute event emissions pose a danger to integrators who may wish to determine vault yield through observation of these events. Additionally, withdrawal requests containing the remaining vault shares can revert due to arithmetic underflow.

Recommendation

Perform the conversion calculation before the rates are changed by the `_burn()` function.

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#4.

3. Medium - Incorrect `_burn` function in `cancelWithdrawalRequest()` (engn33r)

When the `Redistribute` error is emitted, it signals that value of a vault share has increased because a staker has effectively donated shares to the vault. But when `_burn()` with two function arguments is called, the key values `issuanceRate` and `freeAssets` are not updated like they are when `_burn()` with

five function arguments is called. `_burn` with two function arguments is called in `cancelWithdrawalRequest()` when instead the five function arguments `_burn()` should be used.

Technical Details

The standard ERC20 `_burn()` function with two function arguments exists in the contracts inherited from the ERC20 contract, but RevenueDistributionToken has a `_burn()` function with five function arguments. The latter is what is normally called in LockedRevenueDistributionToken (1, 2, 3) except for in `cancelWithdrawalRequest()`. The purpose of line 137 is to burn any excess shares beyond what is returned to the staker cancelling their withdrawal request, because in this case the staker will receive the same value of shares (priced in underlying assets) as when they deposited, but this may be a lesser number of shares than they submitted a withdrawal request for. Phrased another way, the accrued value that the shares gained while the contract held the shares is returned to the vault to be shared among shareholders. When the value in the vault has changed, the value of each share should change by updating `freeAssets` and `issuanceRate`. But `freeAssets` and `issuanceRate` are not updated when `_burn()` with two function arguments is called. So after `cancelWithdrawalRequest()` is called, `freeAssets` and `issuanceRate` may not be accurate. Specifically, `freeAssets` could be overinflated by `convertToAssets(burnShares_)`. And if the `_burn` happens after `vestingPeriodFinish`, the `issuanceRate` will not be updated to zero, meaning the return value of `totalAssets()` would not be accurate.

Impact

Medium. State variables do not get updated properly because incorrect function is called.

Recommendation

Modify this line of `cancelWithdrawalRequest()`. The caller should be `msg.sender` for correctness but keeping `caller == owner` avoids the need to set allowances, which saves gas and improves the user experience.

```
- _burn(address(this), burnShares_);
+ _burn(burnShares_, convertToAssets(burnShares_), address(this), address(this),
address(this));
```

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#1.

4. Medium - `updateVestingSchedule()` may not be called on time (engn33r)

Stakers are incentivized to call `updateVestingSchedule()` when the vestingPeriodFinish is nearly over. However, stakers may not run keepers or bots to call this function because this is no reward to the caller of this function other than the standard staking rewards shared with the pool. This could lead to a failure to call `updateVestingSchedule()` before `vestingPeriodFinish`, resulting in stakers losing out on rewards.

Technical Details

There are two closely related issues here:

1. `updateVestingSchedule()` can be called any time within 24 hours of `vestingPeriodFinish`. The optimal strategy for calling `updateVestingSchedule()` from the perspective of a staker who will withdraw from the vault if the yield is any lower is to 1. call `updateVestingSchedule()` as soon as possible if `issuanceRate` will increase from its current value (the rate of rewards distribution increases) or 2. delay calling `updateVestingSchedule()` until `vestingPeriodFinish` if `issuanceRate` will decrease. The issue here is that by delaying the `updateVestingSchedule()` call, the function may not be called until after `vestingPeriodFinish`. It may be even further delayed if gas is very high and the staker(s) running bots to automatically call the function determine that the “lost yield” from the time where `issuanceRate` is zero is less than the gas cost of calling `vestingPeriodFinish`. This is possible because the staker(s) running bots may not have a large number of vault shares.
2. `issuanceRate` can only be changed in `updateVestingSchedule()` except when `block.timestamp > vestingPeriodFinish`. When `block.timestamp > vestingPeriodFinish`, `issuanceRate` can be set to zero when the `_burn()` or `_mint()` RevenueDistributionToken functions are called. This could be done accidentally, with a normal user wishing to stake into the vault who unknowingly sets `issuanceRate` to zero while staking their GET. Even if `issuanceRate` is non-zero, rewards will not be distributed after `vestingPeriodFinish` due to the math in `totalAssets()`. Any time that `vestingPeriodFinish` is less than the current block.timestamp is a zero yield

scenario for stakers.

In short, an `issuanceRate` of zero is not good for stakers, but there is a question of who will call `updateVestingSchedule()` and when they will call it. Competing incentives and a lack of integration with a keeper system like Gelato may result in non-ideal scenarios for all stakers.

Impact

Medium. Any user can prevent this scenario from happening, but there is a cost to preventing the scenario that may not be reimbursed sufficiently to incentivize the avoidance an `issuanceRate` of zero for a temporary period of time.

Recommendation

Integrate with a keeper solution like Gelato or provide a reward for the caller of `updateVestingSchedule()`. Another option is for the GET team operate a system to automatically call `updateVestingSchedule()` when a valid time arises. The GET team should create monitoring to track how close to `vestingPeriodFinish` the `updateVestingSchedule()` function gets called, with an additional alert if `issuanceRate` ever reaches zero.

Developer Response

Acknowledged, will be mitigated.

The GET Protocol DAO plan to operate keepers to ensure that `updateVestingSchedule()` will be called on a regular basis.

5. Medium - External call made to non-existent permit function (securerodd)

The LockedRevenueDistributionToken.sol contract makes external calls to a contract containing calldata for a function that does not exist on that contract.

Technical Details

The `mintWithPermit()` and `depositWithPermit()` functions are intended to allow users to transfer assets to the vault in a single transaction. To achieve this, these functions call permit on the underlying asset with a user's signature that was gathered off-chain. However, the vault's underlying asset is intended to be GET which does not contain a permit function on Ethereum or Polygon.

Impact
Medium. Currently, transactions that call either of these permit functions will revert without a helpful error message.

Recommendation
If the codebase is not intended to be reused for vaults that hold different underlying assets, consider removing the defunct `mintWithPermit()` and `depositWithPermit()` functions.

Developer Response
Acknowledged, won't fix.

While the GET token does not implement a permit function, the contracts have been written to be generic to ERC20 assets with permit functionality so these functions will remain present for other users of the contracts.

6. Medium - Vault lacks slippage protection (securerodd)
According to documentation, EOA users will be interacting directly with the GovernanceLockedRevenueDistributionToken.sol contract. Key functions for depositing to and withdrawing from the vault do not offer slippage protection for price changes between when an EOA checks the current conversion rate and when they send their deposit or withdrawal transaction.

Technical Details
When an EOA user wants to participate in staking with the vault, they can check what the current exchange rate of their GET and/or xGET will be using functions such as `previewDeposit()`. Since they are an EOA, they must then wait for this transaction to be mined for it to return results before they can post a new transaction acting on this information. There is no guarantee that the previously returned exchange rate will be the same and in fact it could be drastically different. There are numerous reasons a price change may occur, such as from exploitative front-run attempts, reward distributions, or even the ratio completely resetting if all of the shares have been removed from the vault. ERC-4626 alludes to this issue in its security considerations, and many other protocols in the ecosystem that offer swaps and exchanges have slippage protection mechanisms for their users.

Impact
Medium. Users who interact directly with the vault are not protected from price changes and may spend and receive more or less than they anticipate.

Recommendation

Consider either adding extension functions such as those described in ERC-5143 or add a router contract to the protocol for EOAs to interface with that offers slippage protection.

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#5.

Low Findings

1. Low - Unsafe casting (engn33r)

The GovernanceLockedRevenueDistributionToken.sol contract is based on the OpenZeppelin ERC20Votes.sol and ERC20VotesComp.sol contracts. One difference is that the OpenZeppelin contracts use the SafeCast library while GovernanceLockedRevenueDistributionToken.sol does not. This can lead to overflows because casting is not safe like other safe math operations introduced in solidity 0.8.X.

Technical Details

OpenZeppelin uses SafeCast in functions like `numCheckpoints()`, `getCurrentVotes()`, and `getPriorVotes()` to prevent overflows due to casting. GovernanceLockedRevenueDistributionToken.sol has similar implementations of `numCheckpoints()`, `getCurrentVotes()`, and `getPriorVotes()` but without SafeCast. The SafeCast library protects against overflows when casting from a large int value to a small int value, and GovernanceLockedRevenueDistributionToken.sol doesn't have any casting overflow protection in the existing contract.

A comparison of the `numCheckpoints()` implementation is compared below.

```
    function numCheckpoints(address account_) public view virtual override returns
(uint32 numCheckpoints_) {
-        numCheckpoints_ = uint32(userCheckpoints[account_].length); // this is the
GovernanceLockedRevenueDistributionToken.sol implementation
+        numCheckpoints_ =  SafeCast.toUint32(_checkpoints[account].length); //
this is how OpenZeppelin implements the casting
    }
```

Impact

Low. Borrowing code from a well-known implementation but then removing the safety measures does not follow security best practices and at worst can lead to overflow conditions.

Recommendation

Use SafeCast like OpenZeppelin on relevant functions. Compound's COMP uses safe casting too.

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#6.

2. Low - `previewWithdrawalRequest()` may return incorrect user's request (securerodd)

The `previewWithdrawalRequest()` function in the LockedRevenueDistributionToken.sol contract is used to preview the results of executing a particular withdrawal request. It allows the caller to specify a particular request and its owner, however, it always gathers the caller's withdrawal request instead of the specified owner's.

Technical Details

Regardless of what value for owner_ is passed to the `previewWithdrawalRequest()`, the withdrawal request is gathered based on the msg.sender. The owner_ value is then used to determine if this request should be fee exempt or not. This incorrectly associates the withdrawal request and any corresponding fee exemptions. Adding the following POC to InstantWithdrawal.t.sol demonstrates the issue:

```
function testIncorrectPreviewWithdrawal() public {
    // Exempt bob from fees
    vault.setWithdrawalFeeExemption(bob, true);

    // Create a deposit and withdrawal request for 2 ether for Bob
    // This should be fee exempt
    _setUpDepositor(bob, 2 ether);
    vault.createWithdrawalRequest(vault.balanceOf(bob));

    // Create a deposit and withdrawal request for 1 ether for Alice
    // This should not be fee exempt
```

```
    // Note that the current prank is now using alice's context
    _setUpDepositor(alice, 1 ether);
    vault.createWithdrawalRequest(vault.balanceOf(alice));

    // Bob should have no withdrawal fees and a balance of 2 ether
    // so we would expect this to return 2 ether. Instead it returns 1 ether
    // which is a combination of Alice's withdrawal request and bob's fee exemption
    (, uint256 bobWithdrawAmount, uint256 bobFee) =
vault.previewWithdrawalRequest(0, bob);
    console.log("Amount to withdraw: %s", bobWithdrawAmount);
}
```

Impact

Low. Users may receive fewer funds than expected if they anticipated the results of an `executeWithdrawalRequest()` to match the output from `previewWithdrawalRequest()`.

Recommendation

Modify `previewWithdrawalRequest()` to gather the withdrawal request associated with the passed in owner_ as opposed to the msg.sender.

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#7.

3. Low - Withdrawal requests incentivized if `issuanceRate` is zero (engn33r)

If `issuanceRate` is zero, stakers will be incentivized to call `createWithdrawalRequest()` because there is no opportunity cost.

Technical Details

When a withdrawal request is created, the contract takes and holds the staker's shares until the withdrawal is executed or canceled. The staker receives no interest or yield while the contract holds the shares. But if the interest or yield is zero, stakers are incentivized to submit a withdrawal request because there are no rewards to miss out on when the contract holds the shares, which happens when `issuanceRate` is zero.

Impact

Low. While this scenario is very unlikely to happen, if it did happen, it could

effectively lead to a bank run on the vault.

Recommendation

Add tests to confirm that a scenario similar to a bank run does not result in any loss of user funds. Consider adding a monitoring mechanism to notify GET governance when the `issuanceRate` value reaches a value below a certain threshold and is nearing zero.

Developer Response

Acknowledged, will be mitigated.

Monitoring will be added alongside the keeper solution to help ensure this scenario is avoided.

4. Low - Lack of issuance rate synchronization across chains may lead to uneven voting rewards (securerodd)

The GovernanceLockedRevenueDistributionToken.sol contract rewards its stakers over a vesting schedule based on the vault's issuance rate. The distributed rewards are in the underlying asset (GET) which is used to denominate a staker's voting power. The GovernanceLockedRevenueDistributionToken.sol is intended to be deployed to multiple chains, however, there is no guarantee the issuance rate will be synchronized across chains and that staker's voting power will increase proportionally the same on each chain.

Technical Details

Voting power is denominated in the underlying assets (GET) as opposed to the vault's shares (xGET) to better facilitate voting across chains. This way, stakers on a vault with a lower share to underlying asset exchange rate are not necessarily penalized (as far as voting is concerned). This does not necessarily hold true for reward distribution. Rewards are granted through a vesting schedule whereby an `issuanceRate` is calculated to issue rewards linearly throughout the vesting period. Since rewards are issued in GET, higher `issuanceRate` vaults may reward their stakers with more voting power.

Impact

Low. Stakers on certain chains may accrue voting power faster than others. The discrepancy in reward distribution rates may also lead to attempts to game the protocol by allocating and moving funds during opportune

moments. While this impact could be more significant, the development team may be able to influence reward distributions lowering the likelihood of significant issuance rate discrepancies occurring.

Recommendation

Complexities surrounding the current voting mechanism may be reduced by requiring votes to occur on a single chain. If this is not desirable, consider monitoring the `issuanceRate` of each vault and taking steps to keep them closely synchronized.

Developer Response

Acknowledged, won't fix.

Rewards will be approximately balanced between networks but this will have some small variance dependent on deposits and withdrawals. Should there be large discrepencies this can be controlled by the rewards directed to each network.

5. Low - Incorrect value `getVotes(address account)` (spalen)

The function `getVotes(address account)` returns the incorrect value because of additional calculation.

Technical Details

The function is changed from standard one to include additional votes that will be available after the next action from the user. Only after the user write action a new checkpoint is stored for him, with a new vote value.

Impact

Low. By providing the future value instead of the real one, the user can get false information about his voting power. It is a problem because a user must take additional action to get voting power as provided.

Recommendation

Provide get votes as in standard implementation. Expose an additional function to provide votes with additional pending votes.

```
function getVotes(address account_) public view virtual override returns (uint256
votes_) {
    uint256 pos_ = userCheckpoints[account_].length;
    if (pos_ == 0) {
```

```
        return 0;
    }
    unchecked {
        votes_ = userCheckpoints[account_][pos_ - 1].shares;
    }
}
```

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#8.

6. Low - Lack of input sanitization in the constructor function (shung)

The contract can be initialized with bad values, which can cause denial of service (DoS).

Technical Details

The variables `instantWithdrawalFee` and `lockTime` are not checked to be valid in the constructor.

```
    constructor(
        string memory name_,
        string memory symbol_,
        address owner_,
        address asset_,
        uint256 precision_,
        uint256 instantWithdrawalFee_,
        uint256 lockTime_
    )
        RevenueDistributionToken(name_, symbol_, owner_, asset_, precision_)
    {
        instantWithdrawalFee = instantWithdrawalFee_;
        lockTime = lockTime_;
    }
```

LockedRevenueDistributionToken.sol#L55-L56

Impact

Low. Core functions might not work until the variables are re-set to

appropriate values by the admin.

Recommendation

Consider applying the following diff which will ensure input sanitization during deployment.

```
diff --git a/src/LockedRevenueDistributionToken.sol
b/src/LockedRevenueDistributionToken.sol
index 512e681..1ea0db8 100644
--- a/src/LockedRevenueDistributionToken.sol
+++ b/src/LockedRevenueDistributionToken.sol
@@ -52,8 +52,8 @@ contract LockedRevenueDistributionToken is
ILockedRevenueDistributionToken, Reve
     )
         RevenueDistributionToken(name_, symbol_, owner_, asset_, precision_)
     {
-        instantWithdrawalFee = instantWithdrawalFee_;
-        lockTime = lockTime_;
+        instantWithdrawalFee = setInstantWithdrawalFee(instantWithdrawalFee_);
+        lockTime = setLockTime(lockTime_);
     }

     /*
@@ -63,7 +63,7 @@ contract LockedRevenueDistributionToken is
ILockedRevenueDistributionToken, Reve
     /**
      * @inheritdoc ILockedRevenueDistributionToken
      */
-    function setInstantWithdrawalFee(uint256 percentage_) external virtual
override {
+    function setInstantWithdrawalFee(uint256 percentage_) public virtual override
{
         require(msg.sender == owner, "LRDT:CALLER_NOT_OWNER");
         require(percentage_ < 100, "LRDT:INVALID_FEE");

@@ -75,7 +75,7 @@ contract LockedRevenueDistributionToken is
ILockedRevenueDistributionToken, Reve
     /**
```

```
     * @inheritdoc ILockedRevenueDistributionToken
     */
-    function setLockTime(uint256 lockTime_) external virtual override {
+    function setLockTime(uint256 lockTime_) public virtual override {
         require(msg.sender == owner, "LRDT:CALLER_NOT_OWNER");
         require(lockTime_ <= MAXIMUM_LOCK_TIME, "LRDT:INVALID_LOCK_TIME");
```

Developer Response

Acknowledged, won't fix.

The recommended fix would have required the deployer to also be the owner due to the ownership check in the setter. As this can be validated after the initial deployment this will be part of the manual deployment testing rather than implemented in code.

7. Low - Add require statement in `updateVestingSchedule` (prady)

As `updateVestingSchedule` is a public function anyone can call the function, and can lock the vesting period even if the `issuanceRate_` turns out to be zero. To tackle this there exists a `updateVestingSchedule` in parent contact `RevenueDistributionToken` which can be only called by `owner`, But this case can be suppressed by adding a require statement.

Technical Details

The function updateVestingSchedule allows anyone to update the vesting schedule even if the `issuanceRate_` is zero.

Impact

Low. As there exists owner function to suppress this case in another call.

Recommendation

This can be added here

```
+ require(issuanceRate_ > 0, "LRDT:UVS:ZERO_ISSUANCE_RATE");
```

Developer Response

Fixed in GETProtocolDAO/locked-revenue-distribution-token#9.

Gas Savings Findings

1. Gas - Avoid && logic in require statements (engn33r)

Using && logic in require statements uses more gas than using separate require statements. Dividing the logic into multiple require statements is more gas efficient.

Technical Details

One instance of require with && logic was found.

Impact

Gas savings.

Recommendation

Replace require statements that use && by dividing up the logic into multiple require statements.

2. Gas - Use Solidity errors in 0.8.4+ (engn33r, PraneshASP)

Using solidity errors is a new and more gas efficient way to revert on failure states.

Technical Details

Require statements are used in the strategy and error messages are not used anywhere. Using this new solidity feature can provide gas savings on revert conditions.

Impact

Gas savings.

Recommendation

In total, there are 19 instances of `require` statements used. Consider changing them to custom errors to save gas.

- 13 instances in `LockedRevenueDistributionToken.sol`
- 6 instances in `GovernanceLockedRevenueDistributionToken.sol`

3. Gas - Use unchecked if no underflow risk (engn33r, spalen)

There is a subtraction operation that can use unchecked for gas savings.

Technical Details

There is at least one location where unchecked can be applied

```
+                   unchecked {
        return
            high_ == 0
            ? 0
            : (isVotes_ ? _unsafeAccess(ckpts, high_ - 1).votes :
_unsafeAccess(ckpts, high_ - 1).shares);
+       }
```

There is another example of a subtraction operation that can use unchecked for gas savings in LockedRevenueDistributionToken.

```
if (withdrawalFeeExemptions[owner_] || request_.unlockedAt <= block.timestamp) {
    return (request_, request_.assets, 0);
}

- uint256 remainingTime_ = request_.unlockedAt - block.timestamp;
+ uint256 remainingTime_;
+ unchecked { remainingTime_ = request_.unlockedAt - block.timestamp; }
```

Impact

Gas savings.

Recommendation

Use unchecked if there is no overflow or underflow risk for gas savings.

4. Gas - Make `DELEGATE_TYPEHASH` private (engn33r)

`DELEGATE_TYPEHASH` is a private constant in the OpenZeppelin implementation but is public in the GET implementation.

Technical Details

`DELEGATE_TYPEHASH` is a private constant in the OpenZeppelin contract that the GET contract is based on. This variable is public in the GET GovernanceLockedRevenueDistributionToken.sol contract. Note this has been applied to `PERMIT_TYPEHASH` already, which is public in OpenZeppelin but private in GET.

Impact
Gas savings.

Recommendation
Use private variables instead of public when possible. If `DELEGATE_TYPEHASH` is made private, remove `DELEGATE_TYPEHASH()` from IGovernanceLockedRevenueDistributionToken.sol.

5. Gas - Duplicate zero check (engn33r)
The `_assets` zero check in `_mint()` and `_burn()` can be removed because it is redundant.

Technical Details
`_mint()` is always called with the argument `shares_ = previewDeposit(assets_)` or with `assets_ = previewMint(shares_)`. `previewDeposit()` rounds down while `previewMint()` rounds up. This means if `assets_` is zero, `shares_` will be zero, but it is possible for `shares_` to be zero when `assets_` is non-zero. Therefore the weaker check of `assets_` can be removed. The same change can be made to `_burn()`.

Impact
Gas savings.

Recommendation
Remove this line and this line checking if `assets_` is zero.

6. Gas - Duplicate balance check (engn33r)
When `createWithdrawalRequest()` is called, `balanceOf[msg.sender]` is compared with the shares function argument. This check isn't required because the transfer will revert if `msg.sender` has insufficient shares.

Technical Details
`createWithdrawalRequest()` checks if `msg.sender` has sufficient shares balance, but the ERC20 transfer a few lines later effectively does the same check, because this subtraction would revert if `msg.sender` has insufficient shares balance.

Impact
Gas savings.

Recommendation

Remove this line to save gas on every valid `createWithdrawalRequest()` call, at the cost of slightly more gas spent in cases where the revert happens.

7. Gas - Make variables immutable (engn33r, pandadefi)

Some state variables that cannot be changed outside of the constructor can be immutable.

Technical Details

The `asset` state variable in the constructor of RevenueDistributionToken can be declared immutable. In addition, ERC20.sol `symbol` and `name` aren't changed after the contract is initialized so they can be immutable.

Impact

Gas savings.

Recommendation

Make variables immutable where possible.

8. Gas - Make functions external (engn33r)

If a function is not called internally, it can be external instead of public.

Technical Details

`getPastTotalSupply()` is never called in GovernanceLockedRevenueDistributionToken and can be external instead of public. The change could be applied to `delegate()` too.

Impact

Gas savings.

Recommendation

Declare functions external where possible.

9. Gas - Make variables uint32 (engn33r)

Some variables variables can have uint32 type for gas savings to pack into the same slot.

Technical Details

`instantWithdrawalFee` and `lockTime` have hard coded maximum values of `100` and `MAXIMUM_LOCK_TIME` (or 62899200) respectively. These variables are of uint256 type but can be defined as uint32.

The same approach can be taken with the constant values MAXIMUM_LOCK_TIME, VESTING_PERIOD, and WITHDRAWAL_WINDOW in the same contract.

Impact

Gas savings.

Recommendation

Pack variables to save gas.

10. Gas - Transfer isn't necessary before calling `_burn()` (pandadefi)

`LockedRevenueDistributionToken.sol` transfer funds to the user before burning the funds. It's possible to burn the user owned funds directly from the contract to save a transfer. A valid reason to transfer before burning is to maintain a more accurate event `emit Withdraw(caller_, receiver_, owner_, assets_, shares_);`.

Technical Details

Burn can be performed without the transfer:

```
- _transfer(address(this), msg.sender, request_.shares);
- _burn(request_.shares, assets_, msg.sender, msg.sender, msg.sender);
+ _burn(request_.shares, assets_, msg.sender, address(this), address(this));
```

LockedRevenueDistributionToken.sol#L159

Impact

Gas savings.

Recommendation

Modify `_burn` for it to use shares from the contract.

11. Gas - Unnecessary nonReentrant protection (pandadefi)

In `RevenueDistributionToken.sol` and `LockedRevenueDistributionToken.sol` a non-reentrant check is in place to deposit and withdraw assets. The asset is the GET token deployed on Ethereum mainnet at `0x8a854288a5976036a725879164ca3e91d30c6a1b` by the GET team. There is no risk of a reentrancy call performed by the GET token because the `transfer()` and `transferFrom()` function aren't doing any external call.

Technical Details

The following functions have an unnecessary non-reentrant check:

- RevenueDistributionToken.sol#L99
- RevenueDistributionToken.sol#111
- RevenueDistributionToken.sol#L117
- RevenueDistributionToken.sol#L130
- RevenueDistributionToken.sol#L138
- RevenueDistributionToken.sol#L142
- LockedRevenueDistributionToken.sol#L110
- LockedRevenueDistributionToken.sol#L127
- LockedRevenueDistributionToken.sol#L152
- LockedRevenueDistributionToken.sol#L205

Impact

Gas savings.

Recommendation

Remove the nonReentrant modifier.

12. Gas - Replace `!=` with `>` (prady)

Technical Details

In `updateVestingSchedule`, instead of using `!=` in L181 `>` can be used for minor gas savings.

```
- require(totalSupply != 0, "LRDT:UVS:ZERO_SUPPLY");
+ require(totalSupply > 0, "LRDT:UVS:ZERO_SUPPLY");
```

Impact

Gas savings.

Recommendation

Use `>` instead of `!=` wherever required.

13. Gas - Emit memory variables in events instead of storage (prady)

Technical Details

In following functions, memory variables can be emitted in events instead of global.

1. setInstantWithdrawalFee

```
- emit InstantWithdrawalFeeChanged(instantWithdrawalFee);
+ emit InstantWithdrawalFeeChanged(percentage_);
```

2. setLockTime

```
- emit LockTimeChanged(lockTime);
+ emit LockTimeChanged(lockTime_);
```

Impact
Gas savings.

Recommendation
Use memory variables while emitting events, instead of storage variables.

14. Gas - Usage of booleans for storage incurs overhead (PraneshASP)

Technical Details
In the `LockedRevenueDistributionToken.sol` file, the `withdrawalFeeExemptions` mapping maps user address with a boolean flag to indicate fee exemption, which could cost some additional gas.

In solidity, booleans (bool) are a bit expensive to use in comparison to other data types because each bool value is stored in a single byte. This means that each bool value requires at least one gas unit to be read or written. In comparison, other data types like uint (unsigned integer) can be stored in much less space, allowing for more efficient use of gas. It’s generally more efficient to use other data types, like uint, whenever possible.

Read more here: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/58f635312aa21f947cae5f8578638a85aa2519f5/contracts/security/ReentrancyGuard.sol#L23-L27

Impact
Gas savings.

Recommendation

Consider using `uint256(1)` for `true` and `uint256(2)` for `false` to reduce gas cost.

withdrawalFeeExemptions

```
- mapping(address => bool) public override withdrawalFeeExemptions;
+ /// @dev uint256(1) indicates true and uint256(2) indicates false
+ mapping(address => uint256) public override withdrawalFeeExemptions;
```

Informational Findings

1. Informational - Redundant imports can be removed (engn33r)

ERC20.sol is imported into GovernanceLockedRevenueDistributionToken three times. The redundant imports can be removed.

Technical Details

GovernanceLockedRevenueDistributionToken.sol imports ERC20.sol. But the ERC20.sol import already exists in LockedRevenueDistributionToken.sol and in RevenueDistributionToken.sol. The ERC20.sol import can be removed from GovernanceLockedRevenueDistributionToken.sol and from LockedRevenueDistributionToken.sol because it will be included from RevenueDistributionToken.sol.

Impact

Informational.

Recommendation

Remove ERC20.sol import from GovernanceLockedRevenueDistributionToken.sol and from LockedRevenueDistributionToken.sol.

2. Informational - Clarify order of operations (engn33r)

Parenthesis can be added to the Math `sqrt()` function to make the order of operations extra clear.

Technical Details

The Math library is based on OpenZeppelin's Math library. The OpenZeppelin library has parenthesis on these lines but the GET Math library doesn't have

these. No issue is expected here, but clarifying the order of operations can make the code easier for readers to understand and make upgrades to future version of solidity less problematic in case the compiler has changes made to the order of operations.

Impact

Informational.

Recommendation

Add parenthesis to clarify order of operations in Math library `sqrt()`.

3. Informational - Operational order during minting could be improved (securerodd, engn33r)

During a single deposit transaction, users are minted shares and therefore given voting power before they actually transfer the underlying asset into the vault. If the underlying token were to contain hooks that allow the token sender to execute code before the transfer (e.g., ERC777 standard), the vault would be giving up control flow in an improper state.

Technical Details

When a user makes a deposit call in the RevenueDistributionToken.sol contract, the underlying `_mint()` function with 4 parameters is called. Inside of this function, the user is given their shares before they officially transfer in the underlying asset to the vault. If reentrancy were to occur in a before token transfer hook during this transferFrom call, the vault would have issued shares without receiving assets in return at that point in time. Reordering these operations may make it possible to remove the `nonReentrant` modifiers on functions that call `_mint()`.

Impact

Informational.

Recommendation

Reorder the calls to perform the transfer before the shares are minted, such as is done in OpenZeppelin's ERC4626 impelementation. The same approach is taken in solmate's ERC4626 `deposit()` and `mint()` functions.

4. Informational - Abandoned withdrawal shares are irretrievable (engn33r, prady)

When a withdrawal request is created, the contract takes and stores that

amount of shares. But the contract does not technically have ownership over these shares, they are effectively escrowed or in limbo until the user takes further actions. This can mean that a user who creates a withdrawal request but never cancels nor executes their withdrawal request can “trust” the vault with their shares but reclaim them at any future point. This is not likely to benefit the user, but unknown unknowns are always possible.

Technical Details

After `createWithdrawalRequest()` is called, the contract takes possession of the user’s shares. But the contract has no way of redeeming these shares that it holds, and there is no time limit for shares to be considered “abandoned” or “forfeited” if left in the contract. This may be similar to a user transferring their shares to an inaccessible random address, except:

1. The contract holding the shares is the ERC4626 vault contract itself
2. The ERC4626 vault cannot do anything with these shares
3. The user can retrieve these shares if the want, unlike sending them to an irretrievable address

Impact

Informational.

Recommendation

Consider adding a “retrieveAbandonedShares” function for shares that have been in the contract for over a set amount of time. The value of the shares would get redistributed to stakers if abandoned.

5. Informational - Shares escrowed for withdrawal can’t vote (engn33r)

When a withdrawal request is created, the contract takes and stores that amount of shares. While the contract holds these shares, the shares are unable to be used for voting.

Technical Details

The GET Locked Revenue Distribution token contracts hold shares that are waiting to be withdrawn, but these shares are not able to be used for voting because the contract never delegates these shares. Because there is no quorum, this is unlikely to result in altering the outcome of a vote directly, but it would make it easier to potentially swing a vote because the shares

held in escrow during the withdrawal process will not vote.

Impact

Informational.

Recommendation

Assuming that the `owner_` function argument in the GET Locked Revenue Distribution token constructor will be the GET multisig address that will already be depositing a large amount of GET and receiving xGET governance shares, add the call `_delegate(address(this), owner_);` into the GovernanceLockedRevenueDistributionToken constructor. Add a similar call in `acceptOwnership()` so delegated votes are shifted when the owner is changed.

6. Informational - Improve function and variable names (securerodd, engn33r)

Certain function and variable names are confusing and could be improved.

Technical Details

1. `createWithdrawalRequest()` takes a function argument of shares and returns a value in units of assets, but the standard ERC4626 `withdraw()` does the opposite by taking assets as a function argument and returning a value in units of shares. Instead, the function might be better named `createRedeemRequest()` if it is going to take a function argument of shares to match the ERC4626 `redeem()` function.
2. `totalAssets_` found in `previewRedeem()` and `previewWithdraw()` doesn't related to the return value of `totalAssets()`. It might be better named `preFeeAssets_` or `assetsPlusFee_`.
3. The Checkpoint array function argument is named `ckpts`, but this is the same name as the state variable `ckpts`. Consider removing this function argument if the same state variable will always be accessed when `_unsafeAccess()` is called.

Impact

Informational.

Recommendation

Use variable names that make sense in the context where they are used.

7. Informational - Use actual GET token contract for tests

(securerodd, engn33r)

The tests use a simplistic MockERC20 for tests but could use the GET token contract for more realistic tests.

Technical Details

Since the GET token is the ERC20 token that the tests care about, consider using the GET contract code directly. The tests can use a fork of mainnet to use the on-chain contract, or the relevant contract code could be imported. If testing on a fork, consider testing on mainnet and Polygon, the two places where these contracts are planned to be deployed.

Impact

Informational.

Recommendation

Use the proper ERC20 token in tests instead of a mock token for more realistic testing results. Doing so may prevent issues such as M5 from occurring.

8. Informational - Documentation nitpicks (engn33r)

The GET staking documentation page has a few errors or omissions.

Technical Details

1. The Governance section makes no mention of this crucial nuance in the governance logic - that the staker must "delegate to themselves in order to activate checkpoints and have their voting power tracked".
2. The list of state variables that can be controlled by governance votes is either incomplete or omits variables that can be changed by the owner outside of governance votes. Specifically, `setPendingOwner()` and `updateVestingSchedule()` are not in the list.

Impact

Informational.

Recommendation

Update the staking documentation.

9. Informational - Conversion calculations not protected from shadow overflow (securerodd)

Several calculations in the codebase attempted to store the result of

multiplication between two uints into a single uint of the same size before performing a division.

Technical Details

In an expression such as `uint256 D = A * B / C`, Solidity will attempt to store the intermediary result of `A * B` before it performs the division by `C`. If this overflows, then the transaction will revert. The muldiv technique was created to prevent the possibility of an intermediary overflow on calculations that perform both multiplication and division.

Impact

Informational.

Recommendation

For any operations that could potentially overflow in an intermediary state, consider using muldiv such as is done in OpenZeppelin's ERC4626 implementation.

10. Informational - Inaccurate comment (securerodd)

There was a docstring in the codebase that contained an inaccurate comment.

Technical Details

`updateVestingSchedule()` has a misleading comment description and a typo that misspelled "Identical" as "Intentical".

Impact

Informational.

Recommendation

Update the comment docstrings to reflect the function's actions and replace the misspelled word.

11. Informational - Mainnet deposit delay incentivized due to bridging delay (engn33r)

The rewards that will be deposited into the xGET contract on Ethereum mainnet originate from Polygon and are bridged to mainnet. This bridging will not be instant. Depositors on Ethereum may be incentivized to wait for a transaction to bridge GET before staking because adding GET into the vault will create an increase in share price when the `issuanceRate` is next updated.

Technical Details

The GET token contract on Ethereum is where GET can be minted, but there is a GET contract on Polygon that is properly mapped and registered with Polygon. The GET token is bridged using the Polygon PoS Bridge, which Polygon docs describe as having a delay of 20 minutes to 3 hours. When GET is bridged across by the bridging contract, stakers will be able to see that rewards are going to arrive soon to the xGET contract on mainnet. The prospective stakers may delay the start of their staking for the amount of time it takes for the assets to bridge and earn yield elsewhere during that time.

Impact

Informational.

Recommendation

Consider avoiding large changes to the issuanceRate from one epoch to the next to avoid short term stakers.

12. Informational - Influence of blockchain data on xGET rewards (engn33r)

Information the the sales of GET ticketed events is visible on-chain. This data may influence stakers in their decision making.

Technical Details

Because GET tickets are NFTs, the sales of those tickets are visible in real-time. The sales of the tickets may provide insight into the future yield that could be earned by stakers. This is not a security concern per se, but may influence the decision making of stakers to stake sooner or later based on this information and depending on how rewards are transmitted to the contract.

Impact

Informational.

Recommendation

Consider the impact that public knowledge of ticket sales may have on stakers. Yield farmers may stake GET during the sales for large events (possibly at known times of year, such as summer festivals) but withdraw from the pool outside these events. Consider maintaining a steady release of rewards into the vault over long periods of time to avoid short-term yield

farmers gaming the system to maximize their rewards at the cost of longer term stakers.

13. Informational - Partial comments (spalen)

The function `_moveVotingPower(address src_, address dst_, uint256 amount_)` is missing `amount_` param in the comments.

Technical Details

The function is defined in GovernanceLockedRevenueDistributionToken: https://github.com/GETProtocolDAO/locked-revenue-distribution-token/blob/ab272ced94d6bc8cc1ded2664408a3fa7ce67128/src/GovernanceLockedRevenueDistributionToken.sol#L300.

Impact

Informational.

Recommendation

Add comment for the missing param `amount_` to have a complete comments for function.

14. Informational - `issuanceRate_` is scaled up (spalen)

Add the comment that the value of `issuanceRate_` is scaled up.

Technical Details

`issuanceRate_` is scaled up using `precision`.

Impact

Informational. User checking the value would get a wrong value without down scaling.

Recommendation

Add the comment that the value is scaled up in an interface file.

15. Informational - Use named constants instead of magic numbers (shung)

Magic numbers are literal values used in the code. They should be replaced with named constants to be descriptive.

Technical Details

`100` in the following instances should use a constant variable instead:

- LockedRevenueDistributionToken.sol#L68
- LockedRevenueDistributionToken.sol#L265
- LockedRevenueDistributionToken.sol#L292
- LockedRevenueDistributionToken.sol#L135

Impact

Informational.

Recommendation

Replace all instances of `100` with a named constant.

16. Informational - Silence compiler warning for unused variables (prady)

To silence the compiler warning about unused variable, don't use variable with name in function params.

Technical Details

If any function param is not used, do not declare that param with a name in functions maxDeposit, maxMint.

For e.g.,

```
- function maxDeposit(address receiver_)
+ function maxDeposit(address)
```

Impact

Informational, alternative way to silence compiler warning.

Recommendation

Remove unused variables.

17. Informational - Use the latest Solidity version with a stable pragma (PraneshASP)

It is generally recommended to use the latest version of Solidity with a stable pragma rather than a floating `pragma ^0.8.7` statement.

Technical Details

Using a stable pragma ensures that the contract will continue to function as intended even if there are breaking changes in future versions of Solidity. By

using a stable pragma, the contract will be locked to a specific version of Solidity, which can help to prevent unexpected behavior or errors if the contract is compiled with a newer version of the compiler.

Also consider using the latest version of Solidity since it will include the most up-to-date features, enhancements, and bug fixes, which can help to make the contract more efficient and secure.

Impact
Informational.

Recommendation
Use strict and latest solidity version in the pragma statement.

18. Informational - Delete doesn’t reduce the size of the array (pandadefi)

`LockedRevenueDistributionToken.sol` has code that `delete` array entries; deleting an array removes the value stored in the array entry but doesn’t reduce the size of the array. This can make the array expensive to load for other contracts using this data.

Technical Details

```
delete userWithdrawalRequests[msg.sender][pos_];
```

LockedRevenueDistributionToken.sol#L131
(LockedRevenueDistributionToken.sol#L157)[https://github.com
/GETProtocolDAO/locked-revenue-distribution-token/blob
/ab272ced94d6bc8cc1ded2664408a3fa7ce67128
/src/LockedRevenueDistributionToken.sol#L157]

Impact
Informational.

Recommendation
If `pos_` doesn’t correspond to the last entry in the array switch `pos_` with the last entry, then use `userWithdrawalRequests[msg.sender].pop()`

Here is an example:

```
request_ = userWithdrawalRequests[pos_];
userWithdrawalRequests[pos_] = userWithdrawalRequests[array.length - 1];
userWithdrawalRequests.pop();
```

Final remarks

engn33r

The GET Locked Revenue Distribution Token builds on strong foundations, using existing implementations for governance, RevenueDistributionToken, and some imports, but combines them in a unique way. This approach avoid the issue of reimplementing an existing design, yet also introduces potential complications. The approach to the contract holding the user's balance when a withdrawal request is created is a good choice to disincentivize users unstaking (because they do not earn yield when the token is unstaked) and to allow the protocol to maintain control over the unstaked value.

securerodd

Overall, the code is well documented and makes good use of existing libraries and projects. I liked the use of RDT as I feel it helped protect against potential vault entry timing abuse issues, although I do have some trepidations about the complexities multi-chain voting may introduce.

About yAcademy

yAcademy is an ecosystem initiative started by Yearn Finance and its ecosystem partners to bootstrap sustainable and collaborative blockchain security reviews and to nurture aspiring security talent. yAcademy includes a fellowship program, a residents program, and a guest auditor program. In the fellowship program, fellows perform a series of periodic security reviews and presentations during the program. Residents are past fellows who continue to gain experience by performing security reviews of contracts submitted to yAcademy for review (such as this contract). Guest auditors are experts with a track record in the security space who temporarily assist with the review efforts.