yAudit Stryke Lumbridge Review

Review Resources:

- A Google doc with the rebrand details

Auditors:

- Jackson
- HHK

Table of Contents

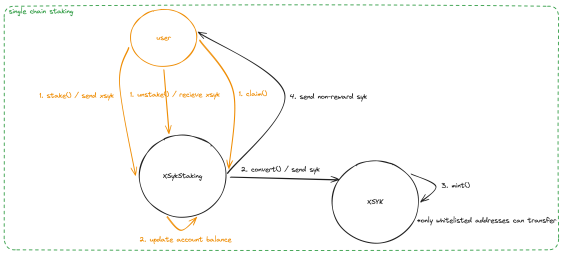
Review Summary

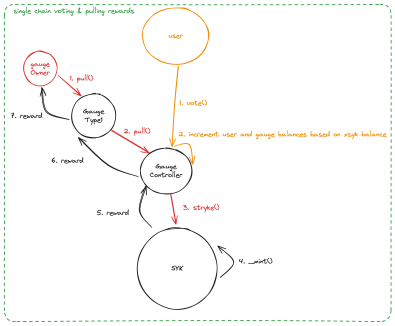
Stryke Lumbridge

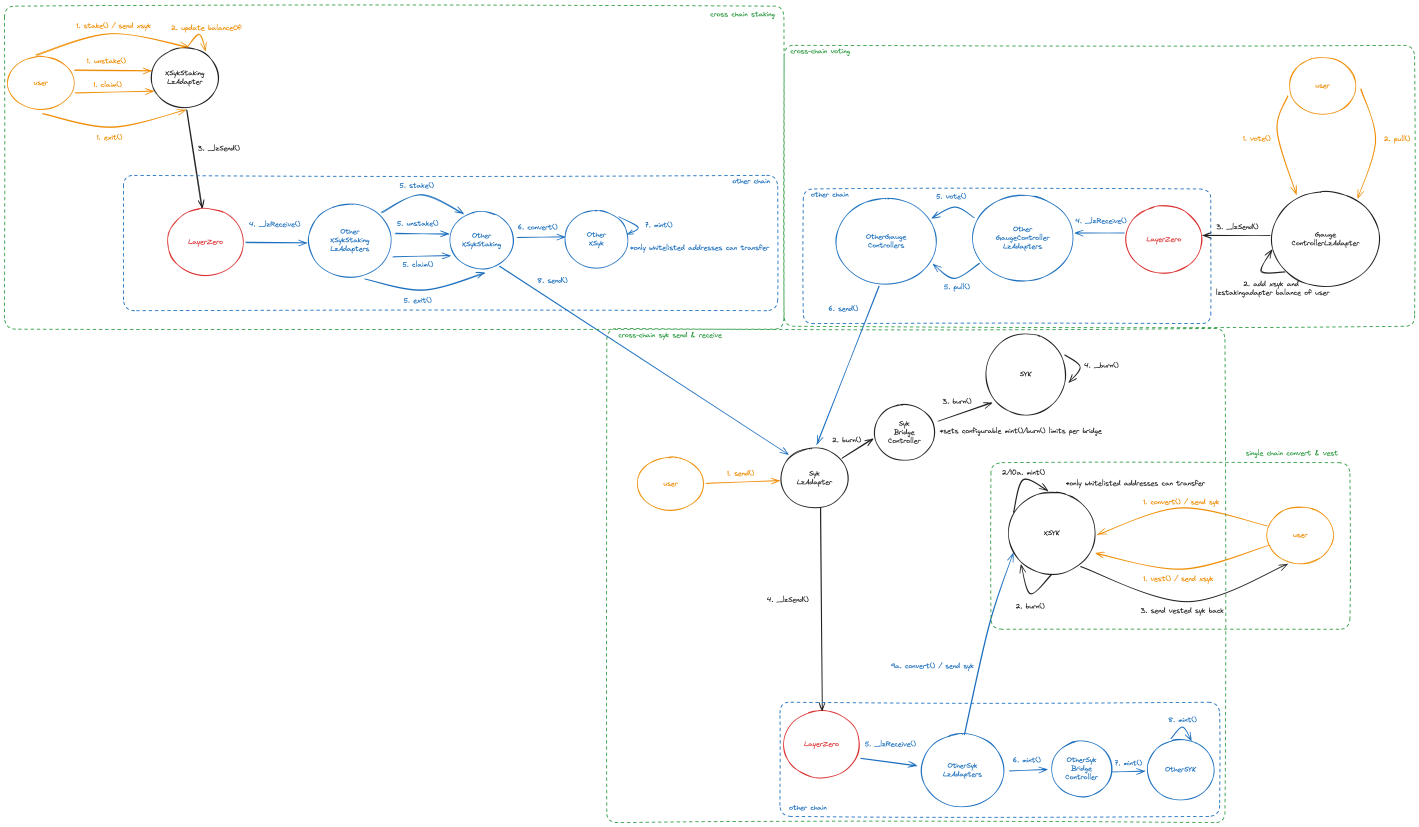
Stryke is a rebrand of the Dopex protocol, which is a decentralized option exchange. The Lumbridge contracts provide a suite of cross-chain contracts: a new token (Syk), a governance system (xSyk), a staking contract to receive protocol fees, a gauge system, and a migrating contract to exchange Dpx and rDpx for Syk.

The contracts of the Stryke Lumbridge Repo were reviewed over 10 days. The code review was performed by 2 auditors between 26-02 and 08-02, 2024. 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 ca38681887734f654db814b4dcb83a958f8056ad for the Stryke Lumbridge repo.

Lumbridge Diagram







Scope

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

```
src
├── gauge
│   ├── GaugeController.sol
│   ├── GaugeType1.sol
│   └── bridge-adapters
│       └── GaugeControllerLzAdapter.sol
├── governance
│   ├── XStrykeToken.sol
│   ├── XSykStaking.sol
│   └── bridge-adapters
│       └── XSykStakingLzAdapter.sol
├── helpers
│   └── ContractWhitelist.sol
├── interfaces
│   ├── IGaugeController.sol
│   ├── IStrykeTokenBase.sol
│   ├── IStrykeTokenRoot.sol
│   ├── ISykBridgeController.sol
│   ├── ISykLzAdapter.sol
│   ├── IXStrykeToken.sol
│   ├── IXSykStaking.sol
│   └── IXSykStakingLzAdapter.sol
├── migration
│   └── SykMigrator.sol
└── token
    ├── StrykeTokenBase.sol
    ├── StrykeTokenChild.sol
    ├── StrykeTokenRoot.sol
    ├── SykBridgeController.sol
    └── bridge-adapters
        └── SykLzAdapter.sol
```

After the findings were presented to the Stryke 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.

yAudit and the auditors make no warranties regarding the security of the code and do not warrant that the code is free from defects. yAudit and the auditors do not represent nor imply to third parties that the code has been audited or is free from defects. By deploying or using the code, Stryke and users of the contracts agree to use the code at their own risk.

Code Evaluation Matrix

Category	Mark	Description
Access Control	Low	The Stryke team has ownership over most contracts, and some of them are upgradeable.
Mathematics	Good	The code doesn't have any complex mathematics.
Complexity	Average	While the code is fairly simple and doesn't have any complicated logic, its cross-chain nature makes it prone to new attack vectors and requires additional mental checks because cross-chain code is not easy to test.
Libraries	Good	The contract only used production-ready libraries such as OpenZeppelin's contracts and LayerZero's OAPP.
Decentralization	Average	Anyone can access and use the different contracts. However, the Stryke team has strong control over the parameters and functioning of the system.
Code stability	Average	No code changes were made during the audit. However, some remediation involved additional complexity, including the addition of a proxy.

Category	Mark	Description
Documentation	Average	No documentation was provided, but most functions have natspec comments.
Monitoring	Average	Events are emitted throughout the contracts. However, there is no indication at this time that there will be protocol-level monitoring to identify unexpected situations caused by the cross-chain nature of the contracts.
Testing and verification	Average	The tests cover most of the code, but some cases are not tested (e.g. Critical issues caused by missing access control). Auditors recommend advanced testing such as fuzzing and invariant testing.

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

1. Critical - Missing access control check allows redeeming and canceling someone else's vesting

Users can convert Syk for xSyk to receive fees from protocol and vote for gauges, to convert back to Syk they have to vest their xSyk for a given period, during this period they can cancel the vesting to get their xSyk back. Once the period ends users can redeem their Syk.

The contract doesn't verify who's canceling or redeeming a vest resulting in allowing anyone to interact with someone else's vesting.

Technical Details

When a user wants to get his Syk tokens back from the xSyk contract he has to vest his tokens for a certain duration by calling the function `vest()`. This function will transfer the xSyk of the user away from him and will start the vesting period.

Once the vesting period ended the user can call `redeem()` to receive his Syk. A user can also cancel his vesting by calling the function `cancelVest()` in this case the xSyk are sent back to the user.

The functions `redeem()` and `cancelVest()` don't have any access control check and assume the `msg.sender` as the creator of the vesting which is an issue as this allows anyone to redeem and cancel vesting of other users as well as receive their tokens.

POC:

`lumbridge/test/XStrykeTokenTest.t.sol`:

```
function test_redeemSomeoneElse() public {
        //JOHN mints 100 xSyk
        syk.mint(john.addr, 100 ether);
        vm.startPrank(john.addr, john.addr);
        syk.approve(address(xSyk), 100 ether);
        xSyk.convert(100 ether, john.addr);

        //JOHN start vesting his tokens
        xSyk.vest(xSyk.balanceOf(john.addr), 180 days);
        vm.stopPrank();

        skip(180 days);

        //Check that vest id 0 is john's vest
        (address vestOwner, uint256 sykAmount,,,) = xSyk.vests(0);
        assertEq(vestOwner, john.addr);

        //DOE redeem JOHN tokens for herself
        uint256 balanceBefore = syk.balanceOf(doe.addr);
        vm.prank(doe.addr, doe.addr);
        xSyk.redeem(0);
        assertEq(syk.balanceOf(doe.addr), balanceBefore + sykAmount);
    }
```

Impact

Critical. Allows redeeming and canceling someone's vesting.

Recommendation

Consider adding a modifier to both functions which verifies that the `msg.sender == vest.account`.

Developer Response

Fixed

2. Critical - Crosschain staking allows to unstake the staked amount and vote on gauge twice

The staking contract allows users with xSyk on a different chain to stake and receive protocol fees. But the way it is used allows a user that staked from a different chain to unstake on the staking contract chains as well as on the original chain resulting in sending him twice his xSyk staked amount.

Technical Details

When staking from different chains, users have to call the `stake()` on the XSykStakingLzAdapter contract that will transfer their tokens to itself and will send forward a message to the chain that the official staking contract is on (Arbitrum).

On this chain, the bridge adapter receives the message and calls the official staking contract's `stake()` functions. This contract takes 2 types of users and acts differently depending on which kind is calling it.

- If the calling user is from the same chain (Arbitrum), then it transfers the xSyk tokens from him and updates its internal balance for the user with the staked amount.
- If the calling user is from a different chain, then the `msg.sender` is the bridge contract, additionally the xSyk was already transferred from the user earlier on the other chain to the bridge contract. Thus, the contract will not try to transfer the xSyk tokens and will just update its internal balance for the user with the supposed staked amount.

When it's time to unstake from the contract, similar to the staking part users can call from the bridge on a different chain, the bridge transfers back tokens that were deposited on it earlier and then forwards a message to the official contract. Or if on the same chain, they call the official staking contract that will transfer back the tokens.

There is an issue with the accounting of this architecture, It updates the balance of a user on the staking contract with no difference if they deposited from another chain or on the main chain (Arbitrum). This results in users being able to unstake directly on the main chain even tho they staked on a different chain. This is an issue as the xSyk balance on the main staking contract will be less than the total staked balance considering the xSyk from other chains are kept on the bridges of each of these chains. The contract could run out of xSyk and users that staked on the main chain will not be able to withdraw.

Additionally, the message design patterns chosen were AB and not ABA which means that the bridge assumes that the transaction will not revert on the destination chain. So even if the transaction on the destination chain reverts the bridge transfers back the xSyk. This results in

users being able to unstake on the main chain and then unstake on the other chain they used originally and thus doubles their unstaking balance.

POC:

`test/IntegrationTest.t.sol`

```
function test_stakingCrossChain_attack() public {
        xSykStaking.setRewardsDuration(7 days);
        // Send 30% of rewards in xSYK
        xSykStaking.updateXSykRewardPercentage(50);
        xSykRoot.updateContractWhitelist(address(xSykStaking), true);

        uint256 amount = 700 ether;

        sykRoot.mint(address(xSykStaking), amount);
        xSykStaking.notifyRewardAmount(amount);

        // Mint syk for Doe on Arbitrum
        sykRoot.mint(doe.addr, 1 ether);
        // Doe stakes from Arbitrum
        vm.startPrank(doe.addr, doe.addr);
        sykRoot.approve(address(xSykRoot), 1 ether);
        xSykRoot.convert(1 ether, doe.addr);
        xSykRoot.approve(address(xSykStaking), 1 ether);
        xSykStaking.stake(1 ether, doe.addr);
        assertEq(xSykStaking.balanceOf(doe.addr), 1 ether);
        vm.stopPrank();

        // Now that our innocent user (Doe) staked on arbitrum we can do the double
withdraw using his balance

        // Mint SYK to john on bsc
        sykBsc.mint(john.addr, 1 ether);

        // John stakes from BSC
        vm.startPrank(john.addr, john.addr);
        sykBsc.approve(address(xSykBsc), 1 ether);
        xSykBsc.convert(1 ether, john.addr);
        bytes memory options =
OptionsBuilder.newOptions().addExecutorLzReceiveOption(200000, 0);
        xSykBsc.approve(address(xSykStakingLzAdapterBsc), 1 ether);
```

```
        xSykStakingLzAdapterBsc.stake{value: 1 ether}(1 ether, options);
        assertEq(xSykStaking.balanceOf(john.addr), 1 ether);
        vm.stopPrank();

        // John can now unstake twice, once on Arbitrum and once on BSC

        // John unstakes from arbitrum
        vm.startPrank(john.addr, john.addr);
        xSykStaking.unstake(1 ether, john.addr);
        assertEq(xSykRoot.balanceOf(john.addr), 1 ether);
        assertEq(xSykStaking.balanceOf(john.addr), 0);
        vm.stopPrank();

        xSykStakingLzAdapterRoot.updateXSykRewardPercentage(50);
        xSykBsc.updateContractWhitelist(address(sykLzAdapterBsc), true);

        // John also unstakes from BSC
        vm.startPrank(john.addr, john.addr);
        options = OptionsBuilder.newOptions().addExecutorLzReceiveOption(200000, 0);
        xSykStakingLzAdapterBsc.unstake{value: 1 ether}(1 ether, options);
        assertEq(xSykBsc.balanceOf(john.addr), 1 ether);
        assertEq(xSykStaking.balanceOf(john.addr), 0);
        vm.stopPrank();
    }
```

Additionally, the gauge controller uses the staking balance of `msg.sender` to determine his power. Saving the balance of the user staking from another chain allows the staker to vote on his original chain and the main staking chain.

Impact

Critical. A user can withdraw on the Arbitrum chain and on the chain he initially staked from doubling his balance. He can also vote twice on the Gauge controller.

Recommendation

- Consider having the bridge pass a hash of the account and chain ID as the staking account so the balance on the staking contract can only be unstaked/voted from the initial chain, similar to the cross-chain gauge.
- Consider implementing an ABA pattern or something similar to ensure transactions didn't revert on the destination chain.

Developer Response

Fixed

High Findings

1. High - Malicious users can delay their cross-chain votes

The way the cross-chain votes are implemented for the gauge system can allow a user to cast votes but have them executed on the destination chain later (days or weeks after), this can be an issue as they could sell their xSyk token in the meantime and thus should not be allowed to vote anymore.

Technical Details

When a user has xSyk on another chain than Arbitrum, they can use the `GaugeControllerLzAdapter` to vote for gauges and redirect rewards. They call the function `vote()` which will send a crosschain message through LayerZero to the adapter on the other chain that then calls the `GaugeController`, the adapter passes the `totalPower` and the `power` of the user to use.

Then on the destination chain, the `GaugeController` gets his `vote()` function called, if the caller is a bridge adapter then it uses the parameters sent by the adapter as the current `totalPower` and `power` of the user to use.

The problem is that when sending a message through LayerZero, the transaction on the destination chain can revert for 2 main reasons:

- Gas. If not enough gas is provided the transaction will revert on the destination chain.

- Logic. The contract called on the destination chain can make the transaction revert.

A failed transaction on the destination chain can be retried, by anyone and at any time.

The current `GaugeController` counts the vote received on the current `epoch`. Knowing this a malicious user can have his tx revert either by providing not enough gas or by voting multiple times and thus the logic of the `GaugeController` would accept only the first transaction and revert on others as it doesn't allow voting more than `totalPower`.

Then at the next `epoch` a user could retry one of the transactions that failed and vote again, he could keep going as long as he has failed transactions available to retry.

This is a problem because a user could vote 100 times, and have the first one succeed and the other 99 fail. Then he can start vesting his xSyk tokens and every week retry one of the 99 other votes. Allowing him to vote for the next 100 `epoch` even tho he doesn't own xSyk anymore.

Impact

High. A user can delay multiple votes and end up voting for free.

Recommendation

Consider having the adapter on the original chain compute and pass the current `epoch` in the vote message, then on the `GaugeController` if the transaction comes from a bridge adapter verify that the `epoch` passed is the current `epoch` otherwise revert.

Developer Response

Fixed

2. High - Crosschain users can keep earning staking rewards after unstaking

The current staking contract allows users to stake their xSyk on other chains using the staking bridge adapter. But if the unstaking transaction is not forwarded on the destination chain because it reverted then a user could unstake from the bridge while keeping earning rewards.

Technical Details

When a user has xSyk on another chain than Arbitrum, they can use the XSykStakingLzAdapter to interact with the staking contract on Arbitrum. They can call the functions: `stake()`, `unstake()`, `claim()` and `exit()`. Each of them will send a crosschain message through LayerZero to the adapter on the other chain that then call the Staking contract.

When staking the adapter will lock the user's xSyk tokens and release them when the user unstakes. This is to ensure xSyk cannot be vested by the user while he is supposed to be staking.

When sending a message through LayerZero, the transaction on the destination chain can revert for 2 main reasons:

- Gas. If not enough gas is provided the transaction will revert on the destination chain.
- Logic. The contract called on the destination chain can make the transaction revert.

A failed transaction on the destination chain can be retried, by anyone at any time.

The current architecture assumes that when a user unstake on the adapter it will always be executed on the Arbitrum staking, but there can be exceptions as described above, while it seems that a logic issue is unlikely to happen, it is easy for a user to provide less gas than needed on the destination chain which would make the unstaking revert on Arbitrum.

So a user could stake some tokens, then unstake but have the transaction revert on the destination chain by giving less gas than needed. The user gets his xSyk back on the original chain and can start vesting them. The Arbitrum staking assumes that the user is still staking, and thus the user can keep earning and claiming rewards as long as he wants.

If the team has a strong monitoring system that allows them to see this failed transaction, they could then force execute it, so the user stops earning rewards. But this is not very likely and not optimal, as users could spam such actions and the team would have to consistently force execute transactions paying gas instead of the user.

Impact

High. A user could keep earning rewards forever after unstaking until it is found out and his failed transaction is force executed by the team.

Recommendation

- Consider enforcing the gas amount passed for the cross-chain unstaking and ensuring it will never revert because of a logic issue.

When enforcing the gas amount make sure to use the quoting system suggested by Layer Zero, this `quote()` function could be called during the `unstake()` execution to make sure we provide the right amount of gas. Providing not enough gas will make the destination transaction revert while providing too much will make the user lose gas fees as there is no refund on the destination chain.

- Consider creating a strong monitoring system for cross-chain transactions.

Developer Response

Fixed in 5a719f8bcea39c69e5bf57c54cad3373fcb3e95e: added ABA pattern.

Medium Findings

1. Medium - Rewards are pullable in an epoch when a gauge has been removed

Technical Details

When a gauge is removed from the `GaugeController`, its `_gaugeId` in the `gauges` mapping is set to an empty `GaugeInfo` such that it can no longer be used. However, the `gaugePowersPerEpoch` mapping is not set to 0. This means that rewards accrued during the epoch in which a gauge is removed can be pulled after a gauge has been removed.

Impact

Medium. Rewards that should not be pullable are pullable and the removed gauge's leftover power results in fewer rewards for the unremoved gauges.

Recommendation

Subtract the gauge's `gaugePowersPerEpoch` from `totalPowerUsedPerEpoch` and set the `gaugePowersPerEpoch` mapping to 0 for the current epoch when a gauge is removed.

Developer Response

Fixed in commit 4bde49e1c4ce143195570deac99dc3a782bad687.

2. Medium - `totalVoteableRewardPerEpoch` should be a mapping

There is an error in the reward accounting, some elements of the rewards computation are not properly updated per `epoch`.

Technical Details

In the Gauge controller when we add or remove a gauge, the `totalVotableRewardPerEpoch` is updated. This variable represents the amount of rewards dedicated to votes, the higher it is, the more a gauge can earn if it receives a majority of votes.

The issue is that when a Gauge calls the `pull()` function it computes the rewards for the `epoch` given, a gauge can pull its rewards for past `epoch`.

The rewards computation uses the `totalVotableRewardPerEpoch` variable to determine how many rewards should be sent.

If a gauge was added or removed after the epoch increasing or decreasing the `totalVotableRewardPerEpoch` the rewards sent to the gauge will differ from what it should have been initially.

This is especially an issue if a gauge hasn't claimed for a long time and starts claiming for an `epoch` that was a long time ago, the `totalVotableRewardPerEpoch` might be completely different and result in much more or less rewards claimed.

Impact

Medium. The reward accounting could be broken.

Recommendation

Consider creating a new variable named `totalVotableReward` and then making `totalVotableRewardPerEpoch` a mapping that is set at the end of the epoch using the `totalVotableReward` value.

Developer Response

Fixed in commit 4bde49e1c4ce143195570deac99dc3a782bad687.

3. Medium - `GaugeControllerLzAdapter` and `XSykStakingLzAdapter` can receive native tokens, but they can't be withdrawn

Technical Details

`GaugeControllerLzAdapter` and `XSykStakingLzAdapter` have `receive()` and `fallback()` functions to receive gas refunds from LayerZero when sending SYK cross-chain. However, there is no way to retrieve the refunds, as such, they'll be stuck in the contracts.

Impact

Medium.

Recommendation

Add methods that can withdraw ETH from the `GaugeControllerLzAdapter` and `XSykStakingLzAdapter` contracts.

Developer Response

Fixed

Low Findings

1. Low - Can't remove a contract from the whitelist

Technical Details

In the `ContractWhitelist` contract the function `updateContractWhitelist()` allows to add a contract to the whitelist. It seems like it's also expected for it to be able to remove a contract from the whitelist considering it takes a `bool _add` as a parameter.

The issue is that on line 29 a check will make it revert when we try to remove a contract from the whitelist.

This could be an issue if some contracts are deprecated or suffer security vulnerabilities and the team decides to remove them from the whitelist.

Impact

Low. Although contracts added to the whitelist are deemed safe, it should be possible to remove them if needed.

Recommendation

Consider removing the check on line 29 from the function.

Developer Response

Fixed in commit ec2296fe23412d7a85c3e0262ee701a0d842246d3.

2. Low - Incorrect inflation check in `StrykeTokenRoot`

Technical Details

The contract `StrykeTokenRoot` is the main Syk contract that will be deployed on Arbitrum. It has a yearly inflation rate set by the team. The function `stryke()` allows restricted caller to mint new Syk tokens following the inflation rate.

This function calls the internal function `availableSupply()` and makes sure we don't mint more than allowed. However, this contract misses 2 important key points:

- Once we minted tokens using the `stryke()` we don't save the amount and date minted at. It results in being allowed to mint the inflation rate unlimited until we reach `maxSupply`.
- Both functions use `totalSupply()` which doesn't take into account the Syk bridged to other chains and thus the contract will be able to mint over the `maxSupply`.

Impact

Low. While the contract is restricted, an error from the team on the GaugeController reward rate or other authorized contracts to mint new Syk could result in minting more tokens than available.

Recommendation

Consider saving the minted amount and date in storage to know how many tokens have been minted, this will protect from both issues as you can revert if `minted > maxSupply` and change `(block.timestamp - genesis)) * emissionRatePerSecond` for `(block.timestamp - lastMintTimestamp)) * emissionRatePerSecond` if `lastMintTimestamp > genesis` in the `availableSupply()` function.

Developer Response

Fixed in commit ebcd42c8f78bfc5aa909d02d0f91583aa020b086 and d624559d91d85499ac4f5ed2e01be0d95342959f.

3. Low - The user might lose his Syk tokens temporarily when bridging

Technical Details

The current bridging architecture uses a controller named `SykBridgeController` that is in deployed on all chains and is in charge of setting mint and burn daily limits.

Once a limit is reached, the user won't be able to mint or burn on this chain until a day has passed. There is an issue with this architecture, as a chain can receive tokens from multiple different chains and can reach its minting limit before other chains reach their burning limit resulting in the Syk token burned on the original chain but not minted on the destination chain. In this case, users will have to wait until more room is available and retry the LayerZero transaction from their explorer which can affect UX.

Additionally, in case of a very busy bridging happening it's not even sure a user will be able to find room for his transaction and might have to wait days and retry a lot of times before being able to finish bridging.

Consider this example:

- Arbitrum: mint limit = 500k, burn limit = 500k
- BSC: mint limit = 250k, burn limit = 250k
- Polygon: mint limit = 300k, burn limit = 300k

While it's clear that BSC or Polygon could suffer the issue explained above, even chains like Arbitrum could fall into the same case. If all users from BSC and Polygon bridge back to Arbitrum the minting limit will be reached.

Impact

Low. Temporarily loose token and bad UX.

Recommendation

Multiple paths are possible:

- Consider having a higher mint limit, this makes it easy to patch but reduces security as in case of infinite mint or hack the attacker would be able to mint a lot of tokens.
- Consider using an ABA message pattern where tokens would be held on the original chain for x minutes, when the returning message is received the bridge burns the tokens otherwise after x minutes the tokens are claimable by the user.

Developer Response

Acknowledged, we will be fixing this issue in a future version of the SYK BridgeController, as of now since we support only a single bridge we don't see this issue arising.

4. Low - Enforce `minDuration` >= `EPOCH_LENGTH` in `XStrykeToken`

Technical Details

Currently, there are no bounds on the `minDuration` in `XStrykeToken`, which means it can be set to 0. However, this is problematic, since if the `minDuration` is 0, it means previously staked SYK can be redeemed in the same block. This implies that SYK could be flashloaned, converted to xSYK, used in voting, and then redeemed in the same block.

This is unlikely to occur since the `minDuration` is currently set to 7 days in the constructor. However, the `minDuration` could be changed in the future via updateRedeemSettings().

Note that there is a relation between `minRatio`, `minDuration`, `EPOCH_LENGTH`, `totalVoteableRewardPerEpoch`, `gaugePowersPerEpoch`, `totalPowerUsedPerEpoch`, and a gauge's `baseReward` that determines the feasibility of this attack.

Impact

Low. Since the `EPOCH_LENGTH` is a constant, the `minRatio` is set to 50, and `minDuration` is set to 7 days, there is not currently a problem, but `minRatio` and `minDuration` are configurable, and the new values may open the protocol up to attack in the future.

Recommendation

Enforce a `minDuration` >= EPOCH_LENGTH, so it is economically unfeasible to vote in a gauge and then redeem xSyk in the same epoch. The `else` branch in `vest()` can also be removed if `minDuration` is always > 0.

It's also recommended that comments are added to `minRatio`, `minDuration`, `EPOCH_LENGTH`, `totalVoteableRewardPerEpoch`, `gaugePowersPerEpoch`, `totalPowerUsedPerEpoch`, and a gauge's `baseReward` noting the importance of their relation to the safety of the protocol.

Developer Response

Acknowledged. The reason to allow `minDuration` to be 0 is to allow for instant redemption in case we have a future migration for the token.

5. Low - Useless ifs in `XSykStaking`

Technical Details

There are checks in `XSykStaking` that have no effect since an uint256 can never be < 0.

- `notifyRewardAmount()`
- `stake()`
- `unstake()`

In the case of `notifyRewardAmount()`, this may have unintended consequences since the `rewardRate` could inadvertently be set to 0.

Impact

Low.

Recommendation

Remove the checks to save gas, or make them `!= 0` if non-zero values are required.

Developer Response

Fixed in commit ebcd42c8f78bfc5aa909d02d0f91583aa020b086

6. Low - Use `xSykRewardPercentage` from `XSykStaking` contract in the adapter

Technical Details

The `XSykStakingLzAdapter` contract is the LayerZero adapter for the staking, it has a `xSykRewardPercentage` variable that it uses in the function `_lzReceive` to determine how much Syk need to be converted into xSyk.

The `XSykStaking` contract already has this variable, having two variables with the same purpose on two contracts might create an issue if they're not updated at the same time or have different values.

Impact

Low. If variables were to not have the same value, users staking cross-chain could receive different rewards than users on Arbitrum.

Recommendation

Make the `XSykStakingLzAdapter` contract call the `XSykStaking` to get the xSyk percentage.

Developer Response

Fixed in commit ebcd42c8f78bfc5aa909d02d0f91583aa020b086

7. Low - Hardcoded Gas Limit in `XSykStakingLzAdapter` and `GaugeControllerLzAdapter` may lead to unintended reverts

Technical Details

`XSykStakingLzAdapter` and `GaugeControllerLzAdapter` call `send()` on the `SykLzAdapter` to send SYK tokens back to the origin chain when necessary. Currently, the gas limit is configured to 200000. However, this may not be sufficient for some chains.

Impact

Low.

Recommendation

Allow the gas limit to be configurable depending on the origin chain, or remove it since the contract of interaction is known.

Also, note that it's recommended to estimate the gas needed to perform the cross-chain messaging using `_quote()` as is done in the `SykLzAdapter` for sends such that users do not overpay (leading to lost native tokens) or underpay (leading to reverts).

Developer Response

Fixed in commit 5a719f8bcea39c69e5bf57c54cad3373fcb3e95e.

8. Low - Non-upgradeable version of `ReentrancyGuard` used in `XStrykeToken`

The contract `XStrykeToken` is upgradeable but doesn't use the upgradeable version of `ReentrancyGuard`.

Technical Details

OpenZeppelin's `ReentrancyGuard` is used in the `XStrykeToken`, but it doesn't use the upgradeable version. This can be risky as if in future upgrades the team were to change the order of inheritance there could be a storage collision.

OpenZeppelin suggests using the upgradeable version of their contracts.

Impact

Low. This could cause storage collision in future upgrades.

Recommendation

Use the upgradeable version of the `ReentrancyGuard` contract.

Developer Response

Fixed in commit ebcd42c8f78bfc5aa909d02d0f91583aa020b086.

9. Low - Some upgradeable contracts are not initialized in `XStrykeToken`

The `XStrykeToken` doesn't initialize all upgradeable contracts, including the name and symbol of the xSyk token.

Technical Details

When inheriting an upgradeable contract it usually has to be initialized in the `initialize()` function. The current implementation of the `XStrykeToken` doesn't initialize the `ERC20Upgradeable`, `ERC20PausableUpgradeable` and `UUPSUpgradeable`.

While not initializing the `UUPSUpgradeable` doesn't impact the contract, not initializing `ERC20Upgradeable` and `ERC20PausableUpgradeable` can be impactful as they update the contract storage, including the name and symbol of the token.

Impact

Low. The token will have no name and symbol and will need to be upgraded.

Recommendation

Initialize all upgradeable contracts.

Developer Response

Fixed in commit ebcd42c8f78bfc5aa909d02d0f91583aa020b086

10. Low - Changing `duration` doesn't update the bridge's `ratePerSecond`

Technical Details

In the `SykBridgeController` the `duration` is used to compute the `ratePerSecond` for bridges. `ratePerSecond` determines the rate at which the limits for a bridge replenish with time.

The `duration` can be changed by calling the `setDuration()` function, but it won't impact existing bridge's `ratePerSecond`.

The team will have to call the `setLimits()` function for each bridge to update their `ratePerSecond`.

If not done there will be a mismatch in `_getCurrentLimit()` as it uses both `duration` and `ratePerSecond` to determine the current limit. This can result in the wrong limit being returned.

Impact

Low. If `ratePerSecond` is not updated manually then the limit returned might not be accurate.

Recommendation

Consider these 2 options:

- Removing the `ratePerSecond` storage variable and determining it manually using the `duration` every time it's needed.

Or

- Adding a parameter `address[] calldata bridges` to the `setDuration()` function and loop through the addresses and update the `ratePerSecond` for each of them.

Developer Response

Fixed in commit ebcd42c8f78bfc5aa909d02d0f91583aa020b086.

11. Low - Potential division by 0 in `computeRewards()`

Technical Details

In the GaugeController, the function `computeRewards()` is used to compute how many rewards should be distributed to a gauge for a given `epoch`.

There is an issue in this internal function, it divides by `totalPowerUsedPerEpoch[epoch]` but if no one voted during the `epoch` then it will revert as it's a division by 0. This is problematic since a gauge has a `baseReward` which means no minimal vote should be needed for it to receive rewards.

Impact

Low. Unlikely to have 0 votes during an `epoch` but if the case occurs then gauges cannot claim rewards.

Recommendation

Add a check `totalPowerUsedPerEpoch[epoch] > 0` and only return the `baseReward` if it's not bigger than 0.

Developer Response

Fixed in commit ebcd42c8f78bfc5aa909d02d0f91583aa020b086.

Gas Saving Findings

1. Gas - Gas improvements in `XSykStaking`

Technical Details

- In the `claim()` function the storage mapping `bridgeAdapters` is read twice.
- In the `exit()` function the storage mapping `balanceOf` is read twice.
- In the `notifyRewardAmount()` function the storage variables `rewardRate`, `duration` and `finishAt` are read multiple times.
- Consider making each variable of the `RedeemSettings` struct `uint64`, so they're all packed in the same storage slot. Similarly, you could make `maturity` a `uint64` or `uint96` and move it just under `address account`, so it gets packed in the same slot.

Impact

Gas savings.

Recommendation

Consider caching storage variables to save unneeded extra `sload` and updating code with recommendations.

Developer Response

Acknowledged.

2. Gas - Cache storage variables and set `constant` in xStrikeToken

Technical Details

- The `syk` is never modified.
- In `getSykByVestingDuration()` the storage struct `redeemSettings` is read multiple times.
- In `vest()` the storage variable `vestIndex` is read twice (`+=` will read before incrementing).

Impact

Gas savings.

Recommendation

Consider caching storage variables to save unneeded extra `sload` and setting `syk` as a `constant`.

Developer Response

Acknowledged and partially fixed in a6a5dcd447cf41a73f89fd0a444d18e8e26715be

3. Gas - Improve `_duration` check in xStrykeToken

Technical Details

The function `getSykByVestingDuration()` calculates the Syk tokens ratio to be received given the vest duration chosen by the users.

On line 126 there is a check that verifies if the duration given exceeds the max duration in which case the function directly returns the max ratio.

This check could be modified from `>` to `>=` as being equal to the max duration also gives the max ratio and thus would save gas from the second part of the function.

Impact

Gas savings.

Recommendation

Change the check on line 126 from `>` to `>=`.

Developer Response

Fixed

4. Gas - Gas improvement in `XSykStakingLzAdapter`

Technical Details

- In `XSykStakingLzAdapter`'s `stake()` function the balance of `xSyk` is retrieved and checked against the `_amount` passed in. However, `xSyk` contains a similar check in ERC20Upgradeable's _update(), which is called during a _transfer()
- Similarly, `_unstake()` checks that the balance is >= the amount, however, if this were not true, the subtraction on L205 would revert. As such these can also be removed.

- In the `exit()` function the storage mapping balanceOf is read twice.
- In the `unstake()` function the storage mapping `balanceOf` is read to check that we don't unstake more than balance but it would revert with underflow if that was the case later when subtracting.

Impact

Gas savings.

Recommendation

Consider caching storage variables and removing useless checks.

Developer Response

Fixed in a6a5dcd447cf41a73f89fd0a444d18e8e26715be and ee99cf38180ed495295d51034cc21fd22af46920.

5. Gas - Gas improvements in `GaugeController` and `GaugeType1`

Technical Details

In `GaugeController`:

- Some state variables are only set once like `syk`, `xSyk`, `xSykStaking`, and could be set `immutable` or `constant`.
- The `GaugeInfo` struct has 2 variables that could be packed together: `uint8 gaugeType` and `address gaugeAddress`, consider declaring them last in the struct, so the solidity compiler packs them in the same storage slot.
- In `addGauge()` multiple storage variables are read multiple times, consider caching them.
- In `removeGauge()` consider using the keyword `delete` since the struct has only simple types instead of setting all variable to 0.
- In `vote()` consider caching the first `epoch()` call, so we don't need to call it 3 times. Additionally, consider doing `accountPowerUsedPerEpoch[epoch][accountId] = usedPower + _voteParams.power` instead of a `+=` to save an extra `sload`.

In `GaugeType1`:

- `gaugeController` and `syk` are only set once and thus should be `immutable` and set in the `constructor()` or `constant`.

Impact

Gas savings.

Recommendation

Consider updating the code with recommendations.

Developer Response

Opted to do some and skip others due to low usage of the functions.
https://github.com/dopex-io/lumbridge/pull/1/commits/a6a5dcd447cf41a73f89fd0a444d18e8e26715be

6. Gas - Gas improvements in `SykMigrator`

Technical Details

- The variables `dpxConversionRate` and `rdpxConversionRate` could be set as `immutable` or `constant`.

Impact

Gas savings.

Recommendation

Consider updating the code with recommendations.

Developer Response

Fixed

7. Gas - Gas Improvements in `SykBridgeController` and `SykLzAdapter`

Technical Details

In `SykBridgeController`:

- In `_mintWithCaller()` and `_burnWithCaller()` consider passing the `_currentLimit` to the `_useMinterLimits()` and `_useBurnerLimits()` internal functions instead of having them recompute the limit.

In `SykLzAdapter`:

- In `_lzReceive()` consider adding a check `if (sykAmount == 0)`, so we don't credit 0 Syk when we want to only receive xSyk tokens.

Impact

Gas savings.

Recommendation

Consider updating the code with recommendations.

Developer Response

Acknowledged.

1st recommendation, won't be optimized. 2nd recommendation is optimized here: https://github.com/dopex-io/lumbridge/pull/1/commits/a6a5dcd447cf41a73f89fd0a444d18e8e26715be

Informational Findings

1. Informational - Unused mapping in GaugeType1 contract

Technical Details

The contract GaugeType1 has an unused mapping `bridgeAdapters` on line 28.

Impact

Informational.

Recommendation

Remove unused mapping.

Developer Response

Fixed in commit 0aa52d9fe072720570c7f30a1934cb8365e84cf5.

2. Informational - Smart contract wallets won't be compatible with xSyk

Technical Details

The xSyk token contract inherits the `ContractWhitelist` contract which has a function `_isEligibleSender()`. It is used in the `convert()` function to revert when a contract calls it unless it is whitelisted.

The motivation for this is to make building a wrapper on top of xSyk impossible unless vetted by the team or governance, but it has a side effect of making smart contract wallets (e.g. Gnosis Safe) incompatible with xSyk.

Impact

Informational. Smart contract wallets like Gnosis Safe won't be compatible with xSyk.

Recommendation

Consider removing this check.

Developer Response

Acknowledged.

3. Informational - xSyk redemption parameters might differ between chains

Technical Details

Because each chain will have its own deployment of the xSyk contract when updating the redemption settings of one contract by calling the `updateRedeemSettings` other deployments will have to be updated as well, ideally all at the same time.

Impact

Informational.

Recommendation

Ensure all deployments have the same redemption parameters and apply any update to all deployments at the same time.

Developer Response

Acknowledged.

4. Informational - Users can vote and pull for gauges when `genesis` is not set

Technical Details

The `genesis` in the GaugeController contract is a variable that corresponds to the timestamp at which rewards are supposed to start.

It starts at 0 and can be set only once by calling the `setGenesis()` functions.

It is used in the `epoch()` internal functions to know in which `epoch` we're in.

Since it's deployed with a value at 0 then the initial `epoch` will be a very high amount, While it's most likely that the team will set the `genesis` before adding new gauges, it would be safer to disallow users from voting/pulling until it is set.

Impact

Info. Users could vote or pull for the wrong `epoch` if the `genesis` is not set. If a user were to pull for a gauge at the wrong `epoch` then we won't be able to pull rewards again when there are rewards to pull.

Recommendation

Consider reverting in the internal function `epoch()` when `genesis == 0`.

Developer Response

Acknowledged, we will not be adding any gauges before setting genesis.

Final remarks

While the code appears simple due to a lack of complex mathematics, the introduction of LayerZero and its asynchronicity adds additional complexity outside what is present in the code. The security posture of this complexity is further compounded by the fact that some of these asynchronous failures are runtime failures and are therefore difficult to test against to ensure correctness.

For this reason, heavy testnet testing is recommended to ensure that all LayerZero infrastructure failure cases have been encountered, accounted for, and have remediation. It is also recommended that close monitoring is put in place to ensure failure cases are caught and promptly fixed.