Biconomy Hyphen 2.0 contest Findings & Analysis Report

2022-07-25

Table of contents

Overview

About C4

Code4rena (C4) is an open organization consisting of security researchers, auditors, developers, and individuals with domain expertise in smart contracts.

A C4 audit contest is an event in which community participants, referred to as Wardens, review, audit, or analyze smart contract logic in exchange for a bounty provided by sponsoring projects.

During the audit contest outlined in this document, C4 conducted an analysis of the Biconomy Hyphen 2.0 smart contract system written in Solidity. The audit contest took place between March 10—March 16 2022.

Wardens

61 Wardens contributed reports to the Biconomy Hyphen 2.0 contest:

1. WatchPug (jtp and ming)
2. cmichel
3. CertoraInc (egjlmn1, OriDabush, ItayG, and shakedwinder)
4. hickuphh3
5. hyh
6. kyliek
7. gzeon
8. sirhashalot
9. 0xDjango
10. pedroais
11. minhquanym
12. throttle
13. Cantor*Dust (d4rk, thank*you, and technovision99)
14. Ruhum
15. danb
16. Dravee
17. benk10
18. kenta
19. IIIIIII
20. wuwe1
21. cccz
22. PPrieditis
23. 0x1f8b
24. peritoflores
25. defsec

26. catchup
27. JMukesh
28. whilom
29. rfa
30. TerrierLover
31. hagrid
32. saian
33. 0xngndev
34. bitbopper
35. hubble (ksk2345 and shri4net)
36. robee
37. berndartmueller
38. Jujic
39. samruna
40. z3s
41. 0xwags
42. 0xNazgul
43. csanuragjain
44. 0v3rf10w
45. jayjonah8
46. yeOlde
47. XDms
48. cryptphi
49. shenwilly
50. Tomio
51. antonttc
52. oyc_109
53. Kenshin
54. Kiep

This contest was judged by pauliax.

Final report assembled by liveactionllama.

Summary

The C4 analysis yielded an aggregated total of 25 unique vulnerabilities. Of these vulnerabilities, 5 received a risk rating in the category of HIGH severity and 20 received a risk rating in the category of MEDIUM severity.

Additionally, C4 analysis included 39 reports detailing issues with a risk rating of LOW severity or non-critical. There were also 39 reports recommending gas optimizations.

All of the issues presented here are linked back to their original finding.

Scope

The code under review can be found within the C4 Biconomy Hyphen 2.0 contest repository and is composed of 7 smart contracts written in the Solidity programming language and includes 1,621 lines of Solidity code.

Severity Criteria

C4 assesses the severity of disclosed vulnerabilities according to a methodology based on OWASP standards.

Vulnerabilities are divided into three primary risk categories: high, medium, and low/non-critical.

High-level considerations for vulnerabilities span the following key areas when conducting assessments:

- Malicious Input Handling
- Escalation of privileges
- Arithmetic
- Gas use

Further information regarding the severity criteria referenced throughout the submission review process, please refer to the documentation provided on the C4 website.

High Risk Findings (5)

[H-01] Can deposit native token for free and steal funds

Submitted by cmichel, also found by CertoraInc

[LiquidityPool.sol#L151]

The `depositErc20` function allows setting `tokenAddress = NATIVE` and does not throw an error.
No matter the `amount` chosen, the `SafeERC20Upgradeable.safeTransferFrom(IERC20Upgradeable(tokenAddress), sender, address(this), amount);` call will not revert because it performs a low-level call to `NATIVE = 0xEeeeeEeeeEeEeeEeEeEeeEEEeeeeEeeeeeeeEEeE`, which is an EOA, and the low-level calls to EOAs always succeed.
Because the `safe*` version is used, the EOA not returning any data does not revert either.

This allows an attacker to deposit infinite native tokens by not paying anything.
The contract will emit the same `Deposit` event as a real `depositNative` call and the attacker receives the native funds on the other chain.

Recommended Mitigation Steps

Check `tokenAddress != NATIVE` in `depositErc20`.

ankurdubey521 (Biconomy) confirmed and commented:

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) commented:

> Great find, definitely deserves a severity of high.

[H-02] `LiquidityProviders.sol` The share price of the LP can be manipulated and making future liquidityProviders unable to `removeLiquidity()`

Submitted by WatchPug

[LiquidityProviders.sol#L345-L362]

```
function removeLiquidity(uint256 _nftId, uint256 _amount)
    external
    nonReentrant
    onlyValidLpToken(_nftId, _msgSender())
    whenNotPaused
{
    (address _tokenAddress, uint256 nftSuppliedLiquidity, uint256 to
    require(_isSupportedToken(_tokenAddress), "ERR__TOKEN_NOT_SUPPOR

    require(_amount != 0, "ERR__INVALID_AMOUNT");
    require(nftSuppliedLiquidity >= _amount, "ERR__INSUFFICIENT_LIQU
    whiteListPeriodManager.beforeLiquidityRemoval(_msgSender(), _tok
    // Claculate how much shares represent input amount
    uint256 lpSharesForInputAmount = _amount * getTokenPriceInLPShar

    // Calculate rewards accumulated
    uint256 eligibleLiquidity = sharesToTokenAmount(totalNFTShares,
```

LiquidityProviders.sol#L192-L194

```
function sharesToTokenAmount(uint256 _shares, address _tokenAddress)
    return (_shares * totalReserve[_tokenAddress]) / totalSharesMint
}
```

The share price of the liquidity can be manipulated to an extremely low value (1 underlying token worth a huge amount of shares), making it possible for `sharesToTokenAmount(totalNFTShares, _tokenAddress)` to overflow in `removeLiquidity()` and therefore freeze users' funds.

Proof of Concept

1. Alice `addTokenLiquidity()` with `1e8 * 1e18` XYZ on B-Chain, totalSharesMinted == `1e44`;
2. Alice `sendFundsToUser()` and bridge `1e8 * 1e18` XYZ from B-Chain to A-Chain;
3. Alice `depositErc20()` and bridge `1e8 * 1e18` XYZ from A-Chain to B-Chain;
4. Alice `removeLiquidity()` and withdraw `1e8 * 1e18 - 1` XYZ, then: `totalReserve == 1 wei` XYZ, and `totalSharesMinted == 1e26`;
5. Bob `addTokenLiquidity()` with `3.4e7 * 1e18` XYZ;
6. Bob tries to `removeLiquidity()`.

Expected Results: Bob to get back the deposits;

Actual Results: The tx reverted due to overflow at `sharesToTokenAmount()`.

Recommended Mitigation Steps

LiquidityProviders.sol#L280-L292

```
    function _increaseLiquidity(uint256 _nftId, uint256 _amount) interna
        (address token, uint256 totalSuppliedLiquidity, uint256 totalSha

        require(_amount > 0, "ERR__AMOUNT_IS_0");
        whiteListPeriodManager.beforeLiquidityAddition(_msgSender(), tok

        uint256 mintedSharesAmount;
        // Adding liquidity in the pool for the first time
        if (totalReserve[token] == 0) {
            mintedSharesAmount = BASE_DIVISOR * _amount;
        } else {
            mintedSharesAmount = (_amount * totalSharesMinted[token]) /
        }
        ...
```

Consider locking part of the first mint's liquidity to maintain a minimum amount of `totalReserve[token]`, so that the share price can not be easily manipulated.

ankurdubey521 (Biconomy) confirmed

pauliax (judge) commented:

> Great find, with a PoC, deserves a severity of high because it is a valid attack path that does not have hand-wavy hypotheticals.

[H-03] Wrong formula when add fee `incentivePool` can lead to loss of funds.

Submitted by minhquanym, also found by cmichel, hickuphh3, and WatchPug

LiquidityPool.sol#L319-L322

The `getAmountToTransfer` function of `LiquidityPool` updates `incentivePool[tokenAddress]` by adding some fee to it but the formula is wrong and the value of `incentivePool[tokenAddress]` will be divided by `BASE_DIVISOR` (10000000000 each time. After just a few time, the value of `incentivePool[tokenAddress]` will become zero and that amount of `tokenAddress` token will be locked in contract.

Proof of concept

Line 319-322

```
    incentivePool[tokenAddress] = (incentivePool[tokenAddress] + (amount
```

Let `x = incentivePool[tokenAddress]`, `y = amount`, `z = transferFeePerc` and `t = tokenManager.getTokensInfo(tokenAddress).equilibriumFee`. Then that be written as

```
    x = (x + (y * (z - t))) / BASE_DIVISOR;
    x = x / BASE_DIVISOR + (y * (z - t)) / BASE_DIVISOR;
```

Recommended Mitigation Steps

Fix the bug by changing lines 319-322 to:

```
    incentivePool[tokenAddress] += (amount * (transferFeePerc - tokenMar
```

ankurdubey521 (Biconomy) confirmed

pauliax (judge) commented:

> Great find, the wrong order of arithmetic operations deserves a severity of high as it would have serious negative consequences.

[H-04] Deleting `nft Info` can cause users' `nft.unpaidRewards` to be permanently erased

Submitted by WatchPug, also found by 0xDjango and hyh

LiquidityFarming.sol#L229-L253

```
function withdraw(uint256 _nftId, address payable _to) external wher
    address msgSender = _msgSender();
    uint256 nftsStakedLength = nftIdsStaked[msgSender].length;
    uint256 index;
    for (index = 0; index < nftsStakedLength; ++index) {
        if (nftIdsStaked[msgSender][index] == _nftId) {
            break;
        }
    }

    require(index != nftsStakedLength, "ERR__NFT_NOT_STAKED");
    nftIdsStaked[msgSender][index] = nftIdsStaked[msgSender][nftIdsS
    nftIdsStaked[msgSender].pop();

    _sendRewardsForNft(_nftId, _to);
    delete nftInfo[_nftId];

    (address baseToken, , uint256 amount) = lpToken.tokenMetadata(_n
    amount /= liquidityProviders.BASE_DIVISOR();
    totalSharesStaked[baseToken] -= amount;

    lpToken.safeTransferFrom(address(this), msgSender, _nftId);

    emit LogWithdraw(msgSender, baseToken, _nftId, _to);
}
```

LiquidityFarming.sol#L122-L165

```
function _sendRewardsForNft(uint256 _nftId, address payable _to) int
    NFTInfo storage nft = nftInfo[_nftId];
    require(nft.isStaked, "ERR__NFT_NOT_STAKED");

    (address baseToken, , uint256 amount) = lpToken.tokenMetadata(_n
    amount /= liquidityProviders.BASE_DIVISOR();

    PoolInfo memory pool = updatePool(baseToken);
    uint256 pending;
    uint256 amountSent;
    if (amount > 0) {
        pending = ((amount * pool.accTokenPerShare) / ACC_TOKEN_PREC
        if (rewardTokens[baseToken] == NATIVE) {
            uint256 balance = address(this).balance;
            if (pending > balance) {
```

```
                unchecked {
                    nft.unpaidRewards = pending - balance;
                }
                (bool success, ) = _to.call{value: balance}("");
                require(success, "ERR__NATIVE_TRANSFER_FAILED");
                amountSent = balance;
            } else {
                nft.unpaidRewards = 0;
                (bool success, ) = _to.call{value: pending}("");
                require(success, "ERR__NATIVE_TRANSFER_FAILED");
                amountSent = pending;
            }
        } else {
            IERC20Upgradeable rewardToken = IERC20Upgradeable(reward
            uint256 balance = rewardToken.balanceOf(address(this));
            if (pending > balance) {
                unchecked {
                    nft.unpaidRewards = pending - balance;
                }
                amountSent = _sendErc20AndGetSentAmount(rewardToken,
            } else {
                nft.unpaidRewards = 0;
                amountSent = _sendErc20AndGetSentAmount(rewardToken,
            }
        }
    }
    nft.rewardDebt = (amount * pool.accTokenPerShare) / ACC_TOKEN_PR
    emit LogOnReward(_msgSender(), baseToken, amountSent, _to);
}
```

When `withdraw()` is called, `_sendRewardsForNft(_nftId, _to)` will be called to send the rewards.

In `_sendRewardsForNft()`, when `address(this).balance` is insufficient at the moment, `nft.unpaidRewards = pending - balance` will be recorded and the user can get it back at the next time.

However, at L244, the whole `nftInfo` is being deleted, so that `nft.unpaidRewards` will also get erased.

There is no way for the user to get back this `unpaidRewards` anymore.

Recommended Mitigation Steps

Consider adding a new parameter named `force` for `withdraw()`, `require(force || unpaidRewards == 0)` before deleting nftInfo.

ankurdubey521 (Biconomy) confirmed and commented:

> Great catch! Thanks a lot for bringing these up.

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) commented:

> Great find, deserves a severity of high as it may incur in funds lost for the users.

KenzoAgada (warden) commented:

> Shouldn't this be medium severity, as only rewards are lost and not original user funds? As the risk TLDR says -

> ```
> 2 — Med: Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack pat with stated assumptions, but external requirements.
> ```

> ```
> 3 — High: Assets can be stolen/lost/compromised directly (or indirectly if there is a valid attack path that does not have hand-wavy hypotheticals).
> ```

> There are other lost-rewards issues that have been classified as high, this questions pertains to them as well.

0xleastwood (warden) commented:

> I would be inclined to keep this as high risk as it is less about the protocol leaking value and more about rewards being completely wiped and lost forever. I would argue, the user's assets at this point in time DO include all unpaid rewards, so it is perfectly reasonable to treat this as high risk.

pauliax (judge) commented:

> Agree that the boundaries are not very clear, this issue might fall somewhere between Medium and High severities. But my initial thought was similar to that of @0xleastwood, the rewards already belong to the user, and losing them will make the user lose on time and

> other opportunities. Also, this is not a hypothetical attack scenario, but a very real valid execution path, thus I think a high severity is fine here.

[H-05] Users will lose a majority or even all of the rewards when the amount of total shares is too large, due to precision loss

Submitted by WatchPug, also found by hyh

[LiquidityFarming.sol#L265-L291]

```
function getUpdatedAccTokenPerShare(address _baseToken) public view
    uint256 accumulator = 0;
    uint256 lastUpdatedTime = poolInfo[_baseToken].lastRewardTime;
    uint256 counter = block.timestamp;
    uint256 i = rewardRateLog[_baseToken].length - 1;
    while (true) {
        if (lastUpdatedTime >= counter) {
            break;
        }
        unchecked {
            accumulator +=
                rewardRateLog[_baseToken][i].rewardsPerSecond *
                (counter - max(lastUpdatedTime, rewardRateLog[_baseT
        }
        counter = rewardRateLog[_baseToken][i].timestamp;
        if (i == 0) {
            break;
        }
        --i;
    }

    // We know that during all the periods that were included in the
    // the value of totalSharesStaked[_baseToken] would not have cha
    // updates to the pool that happened after the lastUpdatedTime.
    accumulator = (accumulator * ACC_TOKEN_PRECISION) / totalSharesS
    return accumulator + poolInfo[_baseToken].accTokenPerShare;
}
```

[LiquidityProviders.sol#L286-L292]

```
uint256 mintedSharesAmount;
// Adding liquidity in the pool for the first time
if (totalReserve[token] == 0) {
    mintedSharesAmount = BASE_DIVISOR * _amount;
```

```
    } else {
        mintedSharesAmount = (_amount * totalSharesMinted[token]) / tota
    }
```

In `HyphenLiquidityFarming`, the `accTokenPerShare` is calculated based on the total staked shares.

However, as the `mintedSharesAmount` can easily become very large on `LiquidityProviders.sol`, all the users can lose their rewards due to precision loss.

Proof of Concept

Given:

- rewardsPerSecond is `10e18`;
- lastRewardTime is 24 hrs ago;

Then:

1. Alice `addTokenLiquidity()` with `1e8 * 1e18` XYZ on B-Chain, totalSharesMinted == `1e44`;
2. Alice `deposit()` to HyphenLiquidityFarming, totalSharesStaked == `1e44`;
3. 24 hrs later, Alice tries to claim the rewards.

`accumulator = rewardsPerSecond * 24 hours` == 864000e18 == 8.64e23

Expected Results: As the sole staker, Alice should get all the `864000e18` rewards.

Actual Results: Alice received 0 rewards.

That's because when `totalSharesStaked > 1e36`, `accumulator = (accumulator * ACC_TOKEN_PRECISION) / totalSharesStaked[_baseToken];` will be round down to `0`.

When the `totalSharesStaked` is large enough, all users will lose their rewards due to precision loss.

Recommended Mitigation Steps

1. Consider lowering the `BASE_DIVISOR` so that the initial share price can be higher;

2. Consider making `ACC_TOKEN_PRECISION` larger to prevent precision loss;

See also the Recommendation on Issue #139.

ankurdubey521 (Biconomy) confirmed

pauliax (judge) commented:

> Great find, probably deserves a severity of high.

Medium Risk Findings (20)

[M-01] Unsupported tokens cannot be withdrawn

Submitted by cmichel, also found by kyliek, pedroais, and PPrieditis

LiquidityProviders.sol#L273

Supported tokens can be turned off again by calling `TokenManager.removeSupportedToken`.
Users won't be able to withdraw their liquidity anymore because of this check in `removeLiquidity`.

Recommended Mitigation Steps

Consider allowing withdrawals even if the token was unsupported to allow users to reclaim their funds.

ankurdubey521 (Biconomy) acknowledged

pauliax (judge) commented:

> A valid concern, assets not at direct risk.

[M-02] A `pauser` can brick the contracts

Submitted by WatchPug, also found by JMukesh, peritoflores, and whilom

Pausable.sol#L65-L68

```
function renouncePauser() external virtual onlyPauser {
    emit PauserChanged(_pauser, address(0));
    _pauser = address(0);
}
```

A malicious or compromised `pauser` can call `pause()` and `renouncePauser()` to brick the contract and all the funds can be frozen.

Proof of Concept

Given:

- Alice (EOA) is the `pauser` of the contract.
- Alice calls `pause()` ;
- Alice calls `renouncePauser()` ;

As a result, most of the contract's methods are now unavailable, and this cannot be reversed even by the `owner` .

Recommended Mitigation Steps

Consider removing `renouncePauser()` , or requiring the contract not in `paused` mode when `renouncePauser()` .

ankurdubey521 (Biconomy) confirmed and commented:

> Yeah, `changePauser` needs to have an `onlyOwner` modifier instead of `onlyPauser` .

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) commented:

> A valid concern, however, the proposed solution has drawbacks too. If you change from onlyPauser to onlyOwner here, a compromise of the owner account will have devastating consequences while with the current implementation the pauser can still pause the contract independently of an owner. So this is a double-edged sword, it is up to you to decide which way is more acceptable.

[M-03] Incompatibility With Rebasing/Deflationary/Inflationary token

Submitted by Jujic, also found by cmichel, defsec, hagrid, hickuphh3, lllllll, minhquanym, Ruhum, and shenwilly

The scope contracts do not appear to support rebasing/deflationary/inflationary tokens whose balance changes during transfers or over time. The necessary checks include at least verifying the amount of tokens transferred to contracts before and after the actual transfer to infer any fees/interest.

Proof of Concept

TokenManager.sol

Recommended Mitigation Steps

Make sure token vault accounts for any rebasing/inflation/deflation.
Add support in contracts for such tokens before accepting user-supplied tokens.

ankurdubey521 (Biconomy) confirmed

pauliax (judge) commented:

> Grouping all the issues related to the incompatibility with weird ERC20s together and making this a primary issue because it is the most generic.

[M-04] Owners have absolute control over protocol

Submitted by throttle, also found by cccz, cmichel, danb, defsec, hickuphh3, lllllll, pedroais, and Ruhum

LiquidityFarming.sol#L174-L192

Owners have full control over the protocol.

Proof of Concept

Owners have full control over:

- executors who perform token transfers on behalf of the destination chain

- reclaiming / withdrawing any tokens (including reward tokens) held by farming contract
- total upgradeability
- instant parameters change (no timelock)
- 1 step owner change (gold standard is 2-step owner change)

Recommended Mitigation Steps

Make executors decentralized.
Add TimeLock for parameter changes.

ankurdubey521 (Biconomy) acknowledged and commented:

> I agree this is an issue, but in the current iteration of Hyphen it is still a centralized system, therefore there is an implicit trust in the contract owners and executors. A decentralized version of the Hyphen bridge is in the works and will fix these issues.

pauliax (judge) commented:

> I am grouping all the issues related to centralization and owner privilege risks together and making this the primary issue because it is the most generic.

[M-05] Frontrunning of `setPerTokenWalletCap` edge case

Submitted by sirhashalot

The `setPerTokenWalletCap()` function in WhitelistPeriodManager.sol contains a comment stating:

```
Special care must be taken when calling this function
There are no checks for _perTokenWalletCap (since it's onlyOwner), b
Checking this on chain will probably require implementing a bbst, wh
Call the view function getMaxCommunityLpPositon() separately before
```

Even if the manual step of calling the `getMaxCommunityLpPositon()` function is properly performed, it is possible for a user to add liquidity to increase the `maxLp` value in between when the `getMaxCommunityLpPositon()` function is called and when the `setPerTokenWalletCap()` function is called. Because this process is manual, this doesn't need to be bot frontrunning in the same block as when the `setPerTokenWalletCap()`

function is called, but can be cause by poor timing of an innocent unknowing user adding liquidity to the protocol. If this condition occurs, the liquidity provider will have provided more liquidity than the perTokenWalletCap limit, breaking the assumptions for this variable and leading to some denial of service conditions.

This edge situation can impact the `setTotalCap()` function and the "perTokenTotalCap[_token]" state variable as well, but the "perTokenWalletCap[_token]" value would have to be reduced before the "perTokenTotalCap[_token]" value is reduced. The impact to `setTotalCap()` follows the same execution path but adds the additional step of calling the `setTotalCap()` function at the end of the process.

Proof of Concept

1. Owner **calls** `getMaxCommunityLpPositon(_token)` **function** to identify maxLp value to confirm new perTokenWalletCap value is below maxLp value
2. An innocent user adds liquidity to their position without the knowledge that the owner is going to reduce the "perTokenWalletCap[_token]" value soon
3. Owner **calls** `setPerTokenWalletCap()` **function** to reduce "perTokenWalletCap[_token]" value
4. The innocent user has more liquidity than the new "perTokenWalletCap[_token]" value. This means that the user can be in a situation where if they remove x amount of liquidity and attempt to add x liquidity back to their position, the innocent user will be unable to do so. Other functions that rely on the assumption that the largest user deposit is below the "perTokenWalletCap[_token]" value may break due to incorrect assumptions

This edge situation can impact the `setTotalCap()` function and the "perTokenTotalCap[_token]" state variable as well, but the "perTokenWalletCap[_token]" value would have to be reduced before the "perTokenTotalCap[_token]" value is reduced. The impact to `setTotalCap()` follows the same execution path but adds the additional step of calling the `setTotalCap()` function at the end of the process.

Recommended Mitigation Steps

A programmatic solution is the only way to avoid these edge case scenarios, though it will increase gas consumption. To convert the manual calling of `getMaxCommunityLpPositon(_token)` to a programmatic solution, add the following require statement next to the existing require statement of the `setPerTokenWalletCap()` function:

```
require(_perTokenWalletCap <= getMaxCommunityLpPositon(_token),
"ERR_PWC_GT_MCLP");
```

ankurdubey521 (Biconomy) acknowledged

pauliax (judge) commented:

> The concern is valid but I do not think that there is any profit for the attacker, and the impac for the regular users is minimal because this value can be updated anytime again by the owner, so I am hesitating if this should be of medium severity or lower, but because the warden provided a nice and comprehensive description, I will leave this in favor of warden.

[M-06] DoS by gas limit

Submitted by danb, also found by benk10 and pedroais

LiquidityFarming.sol#L220
LiquidityFarming.sol#L233

In `deposit` function it is possible to push to `nftIdsStaked` of anyone, an attacker can deposit too many nfts to another user, and when the user will try to withdraw an nft at the end of the list, they will iterate on the list and revert because of gas limit.

ankurdubey521 (Biconomy) confirmed and commented:

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) decreased severity to Medium and commented:

> A valid concern, but I think it should be of medium severity because the victim can still withdraw NFTs one by one until reaching the necessary index because it breaks inside the loop: LiquidityFarming.sol#L234-L235.

[M-07] Sending tokens close to the maximum will fail and user will lose tokens

Submitted by pedroais, also found by WatchPug

LiquidityPool.sol#L171
LiquidityPool.sol#L273

When a user calls the deposit function the reward amount is calculated and an event is emited with amount+reward as the transfer amount. The function checks amount is smaller than the max amount.

An executor then listens to this event and calls sendFundsToUser with rewards + amount as the amount parameter. This function checks amount+reward is smaller than max amount.

This is a problem because the amount transferred may be in the limit but amount + reward could pass the limit and the executor won't be able to send the transaction. The user will lose the funds. Both checks should be made with the reward or without the reward but the checks should be the same for this not to happen.

Step by step :
Max transfer is set to 50 for token A
Bob transfers 49 tokens, this will pass since 49<50. The reward is calculated in 2 tokens.
The executor then calls sendFundsToUser with 52. This transaction will revert and user will lose their tokens.

This value of amount includes rewards but the previous check didn't include rewards: LiquidityPool.sol#L273.

Recommended Mitigation Steps

Both checks should be made over the same amount = amount + rewards

ankurdubey521 (Biconomy) disputed and commented:

> We handle this issue by setting a slightly larger limit in the transfer config of each token on the destination chain.

pauliax (judge) decreased severity to Medium and commented:

> Even though the sponsor is already aware of and mitigates this issue, it could still be fixed algorithmically to prevent accidental loss of funds. I am leaving this as of medium severity.

[M-08] Incentive Pool can be drained without rebalancing the pool

Submitted by kyliek, also found by Ruhum and WatchPug

LiquidityPool.sol#L149-L173
LiquidityPool.sol#L263-L277

`depositErc20` allows an attacker to specify the destination chain to be the same as the source chain and the receiver account to be the same as the caller account. This enables an attacker to drain the incentive pool without rebalancing the pool back to the equilibrium state.

Proof of Concept

This requires the attacker to have some collateral, to begin with. The profit also depends on how much the attacker has. Assume the attacker has enough assets.

In each chain, when the pool is very deficit (e.g. `currentLiquidity` is much less than `providedLiquidity`), which often mean there's a good amount in the Incentive pool after some high valued transfers, then do the following.

- step 1 : borrow the liquidityDifference amount such that one can get the whole incentivePool.

```
uint256 liquidityDifference = providedLiquidity - currer
if (amount >= liquidityDifference) {
    rewardAmount = incentivePool[tokenAddress];
```

- step 2 : call `depositErc20()` with `toChainId` being the same chain and `receiver` being `msg.sender`.

The executor will call `sendFundsToUser` to msg.sender. Then a rewardAmount, equivalent to the entire incentive pool (up to 10% of the total pool value), will be added to `msg.sender` minus equilibrium fee (~0.01%) and gas fee.

In the end, the pool is back to the deficit state as before, the incentive pool is drained and the exploiter pockets the difference of rewardAmount minus fees.

This attack can be repeated on each deployed chain multiple times whenever the incentive pool is profitable (particularly right after a big transfer).

Recommended Mitigation Steps

- Disallow `toChainId` to be the source chain by validating it in `depositErc20` or in `sendFundsToUser` validate that `fromChainId` is not the same as current chain.

- Require `receiver` is not `msg.sender` in `depositErc20`.

tomarsachin2271 (Biconomy) commented:

> If depositor keeps toChainId same as source chain Id, then executor will not pick this depos transaction on backend as there won't be any mapping for fromChainId => toChainId, so depositor funds will remain in the source chain if he tries to do it and try to drain the incentive pool.

> Although this could happen coz of any bug on the UI, so it's better to handle these situation on contract itself. It will increase a gas though a bit while depositing. Will consider this point though.

ankurdubey521 (Biconomy) confirmed and commented:

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) decreased severity to Medium and commented:

> It is always good to enforce such things on the contract level itself if possible. While there are some precautions, there still exists a hypothetical attack path so I am leaving this as of medium severity.

[M-09] Improper Upper Bound Definition on the Fee

Submitted by defsec, also found by catchup, danb, Dravee, gzeon, hickuphh3, hubble, peritoflores, Ruhum, and throttle

The **equilibriumFee** and **maxFee** does not have any upper or lower bounds. Values that are too large will lead to reversions in several critical functions or the LP user will lost all funds when paying the fee.

Proof of Concept

1. Navigate to the following contract.

TokenManager.sol#L52

2. Owner can identify fee amount. That directly affect to LP management.
   LiquidityPool.sol#L352

3. Here you can see there is no upper bound has been defined.

```
function changeFee(
    address tokenAddress,
    uint256 _equilibriumFee,
    uint256 _maxFee
) external override onlyOwner whenNotPaused {
    require(_equilibriumFee != 0, "Equilibrium Fee cannot be 0")
    require(_maxFee != 0, "Max Fee cannot be 0");
    tokensInfo[tokenAddress].equilibriumFee = _equilibriumFee;
    tokensInfo[tokenAddress].maxFee = _maxFee;
    emit FeeChanged(tokenAddress, tokensInfo[tokenAddress].equil
}
```

Recommended Mitigation Steps

Consider defining upper and lower bounds on the **equilibriumFee** and **maxFee**.

ankurdubey521 (Biconomy) confirmed and commented:

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) commented:

> Valid concern. I am grouping all the issues related to the validation of fee variables together and making this the primary one as it contains the most comprehensive description.

[M-10] Call to non-existing contracts returns success

Submitted by Certoralnc, also found by kenta and wuwe1

LiquidityFarming.sol#L140
LiquidityFarming.sol#L145
LiquidityFarming.sol#L187

Low level calls (call, delegate call and static call) return success if the called contract doesn't exist (not deployed or destructed).

This makes a user be able to send his funds to non-existing addresses.

`LiquidityFarming`

`reclaimTokens` - if the owner calls by accident with a non-existing address he'll lose the funds.

`_sendRewardsForNft` - if the `withdraw` or `extractRewards` will be called with a `to` non-existing address, the funds will be lost. That's because of the call to `_sendRewardsForNft` which contains a low level call to the `to` address.

`sendFundsToUser` - if an executor calls by accident with a non-existing address the funds wil be lost.

`transfer` - if the `transfer` function will be called (by the LiquidityProvidors contract of course) with a non existing address as a receiver, the funds will be lost.

This can be seen here https://github.com/Uniswap/v3-core/blob/main/audits/tob/audit.pdf (report #9) and here https://docs.soliditylang.org/en/develop/control-structures.html#error-handling-assert-require-revert-and-exceptions

ankurdubey521 (Biconomy) confirmed and commented:

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) commented:

> I am hesitating if this should be with the severity of Medium or Low but leaving in favor of wardens this time. I believe checking against empty addresses is not enough, low-level calls return true even for non-empty but not valid addresses. It would be better to use interfaces possible.

[M-11] `LiquidityProviders` : Setting new liquidity pool will break contract

Submitted by cmichel, also found by gzeon

LiquidityProviders.sol#L171

Owners can change the `liquidityPool` variable any time with the `setLiquidityPool` function.
If a liquidity pool was already set and users added liquidity with `addTokenLiquidity` , the tokens are directly transferred to the liquidity pool and not kept in the `LiquidityProviders` contract.

Changing the `liquidityPool` to a different contract will make it impossible for the users to withdraw their liquidity using `removeLiquidity` because the tokens are still in the old `liquidityPool` and cannot be retrieved.

All users will lose their funds.

Recommended Mitigation Steps

Changing the `liquidityPool` requires a sophisticated migration mechanism.
Only allow setting the `liquidityPool` contract once.

ankurdubey521 (Biconomy) acknowledged

pauliax (judge) decreased severity to Medium and commented:

> A valid concern, but I am downgrading this to Medium risk because the funds are not lost forever, the same old liquidityPool can be set again by the owner in such a case.

[M-12] `LiquidityProviders` : Setting new LP token will break contract

Submitted by cmichel, also found by gzeon

LiquidityProviders.sol#L116

Owners can change the `lpToken` variable at any time with the `setLpToken` function.
If an LP token was already set and users added liquidity with `addTokenLiquidity` and were minted a `lpToken` NFT, changing the `lpToken` to a different contract will make it impossible for the users to withdraw their liquidity using `removeLiquidity`.

All users will lose their funds.

Recommended Mitigation Steps

Changing the `lpToken` requires a sophisticated migration mechanism.
Only allow setting the `lpToken` contract once.

ankurdubey521 (Biconomy) acknowledged

pauliax (judge) decreased severity to Medium and commented:

> A valid concern, but I am downgrading this to Medium risk because the funds are not lost forever, the same old lpToken can be set again by the owner in such a case.

[M-13] Improper `tokenGasPrice` design can overcharge user for the gas cost by a huge margin

Submitted by WatchPug, also found by cmichel and hyh

LiquidityPool.sol#L330-L337

```
uint256 totalGasUsed = initialGas - gasleft();
totalGasUsed = totalGasUsed + tokenManager.getTokensInfo(tokenAddres
totalGasUsed = totalGasUsed + baseGas;

uint256 gasFee = totalGasUsed * tokenGasPrice;
gasFeeAccumulatedByToken[tokenAddress] = gasFeeAccumulatedByToken[tc
gasFeeAccumulated[tokenAddress][_msgSender()] = gasFeeAccumulated[tc
amountToTransfer = amount - (transferFeeAmount + gasFee);
```

When the `Executor` calls `sendFundsToUser()`, the `tokenGasPrice` will be used to calculat the gas fee for this transaction and it will be deducted from the transfer amount.

However, since `tokenGasPrice` is `uint256`, the smallest chargeable amount is `1 wei` Toker for `1 gas`. But there are tokens like `WBTC` (decimals = 8) or `USDC` (decimals = 6), for these tokens, even `1 wei` of the token can be worth a lot of gas, if the `tokenGasPrice` is set to `1`, `gasFee` will far more than the actual cost; if it's set to `0`, `gasFee` can only be `0`.

Proof of Concept

Given:

- `baseGas` = 21000
- `tokenGasPrice` for WBTC = `1 wei`
- `transferFeeAmount` = 0
- 1 WBTC = 20,000 MATIC
- Alice send `0.1 WBTC` to Bob's address on Polygon

- `Executor` calls `sendFundsToUser()` with `tokenGasPrice = 1` on Polygon, `totalGasUsed = 42000` and the gas price is `30G wei`, `Executor` paid `0.00126 MATIC` for gas.

```
uint256 gasFee = 42000 * 1;
...
amountToTransfer = 10000000 - (0 + 42000);
```

3. Bob received 0.09958 WBTC, and paid `0.00042 WBTC` for the gas, the gas fee was overcharged by 6666 times.

Recommended Mitigation Steps

Consider changing `tokenGasPrice` to a value with decimals of `18` and it should be used like this:

```
uint256 gasFee = totalGasUsed * tokenGasPrice / 1e18;
```

ankurdubey521 (Biconomy) acknowledged and commented:

> I'm not sure I agree with the recommendation since If a token atom's value exceeds the gas paid for the transaction, it would still be truncated if we send a tokenGasPrice multiplied by 10e18 and divide it in the contract.

> But this is a great catch, I think the bigger issue here is that for certain tokens it is not feasible to charge gas fee on a per transaction basis, we'll have to think about how to mitigate this.

pauliax (judge) decreased severity to Medium and commented:

> The sponsor better knows the design and intentions of the system, and they claim to be dealing with the token atoms on a contract level. However, I would still like to emphasize possible risks with different tokens and decimals. It is a common issue, so I would like to group similar issues together and assign them a severity of Medium.

[M-14] `LiquidityFarming.sol` Unbounded for loops can potentially freeze users' funds in edge cases

Submitted by WatchPug

In the current implementation of `withdraw()`, it calls `_sendRewardsForNft()` at L243 which calls `updatePool()` at L129 which calls `getUpdatedAccTokenPerShare()` at L319.

`getUpdatedAccTokenPerShare()` will loop over `rewardRateLog` to calculate an up to date value of accTokenPerShare.

[LiquidityFarming.sol#L270-L285]

```
    while (true) {
        if (lastUpdatedTime >= counter) {
            break;
        }
        unchecked {
            accumulator +=
                rewardRateLog[_baseToken][i].rewardsPerSecond *
                (counter - max(lastUpdatedTime, rewardRateLog[_baseToken
        }
        counter = rewardRateLog[_baseToken][i].timestamp;
        if (i == 0) {
            break;
        }
        --i;
    }
```

This won't be a problem in the usual cases, however, if there is a baseToken that:

- the `rewardPerSecond` get updated quite frequently;
- the liquidityProviders are inactive (no deposits / withdrawals for a period of time)

Then by the time one of the `liquidityProviders` come to `withdraw()`, the tx may revert due to out-of-gas.

As the `rewardRateLog` is now accumulated to a large size that causes the loop costs more gas than the block gas limit.

There is a really easy fix for this, it will also make the code simpler:

Recommended Mitigation Steps

Consider removing `rewardRateLog` and change `setRewardPerSecond()` to:

```
function setRewardPerSecond(address _baseToken, uint256 _rewardPerSe
    updatePool(baseToken);
    rewardRate[_baseToken] = RewardsPerSecondEntry(_rewardPerSecond,
    emit LogRewardPerSecond(_baseToken, _rewardPerSecond);
}
```

ankurdubey521 (Biconomy) acknowledged

pauliax (judge) decreased severity to Medium and commented:

> A valid concern, but I think it should be of Medium severity: *"Assets not at direct risk, but the function of the protocol or its availability could be impacted, or leak value with a hypothetical attack path with stated assumptions, but external requirements."*

[M-15] `WhitelistPeriodManager` : Improper state handling of exclusion removals

Submitted by hickuphh3, also found by throttle

WhitelistPeriodManager.sol#L178-L184
WhitelistPeriodManager.sol#L115-L125

The `totalLiquidity` and `totalLiquidityByLp` mappings are not updated when an addres is removed from the `isExcludedAddress` mapping. While this affects the enforcement of the cap limits and the `getMaxCommunityLpPositon()` function, the worst impact this has is that the address cannot have liquidity removed / transferred due to subtraction overflow.

In particular, users can be prevented from withdrawing their staked LP tokens from the liquidit farming contract should it become non-excluded.

Proof of Concept

- Assume liquidity farming address `0xA` is excluded
- Bob stakes his LP token
- Liquidity farming contract is no longer to be excluded:
  `setIsExcludedAddressStatus([0xA, false])`

- Bob attempts to withdraw liquidity → reverts because `totalLiquidityByLp[USDC][0xA] = 0`, resulting in subtraction overflow.

```
// insert test case in Withdraw test block of LiquidityFarming.tests
it.only('will brick withdrawals by no longer excluding farming contr
  await farmingContract.deposit(1, bob.address);
  await wlpm.setIsExcludedAddressStatus([farmingContract.address], [
  await farmingContract.connect(bob).withdraw(1, bob.address);
});

// results in
// Error: VM Exception while processing transaction: reverted with p
```

Recommended Mitigation Steps

The simplest way is to prevent exclusion removals.

```
function setIsExcludedAddresses(address[] memory _addresses) externa
  for (uint256 i = 0; i < _addresses.length; ++i) {
    isExcludedAddress[_addresses[i]] = true;
    // emit event
    emit AddressExcluded(_addresses[i]);
  }
}
```

ankurdubey521 (Biconomy) confirmed

pauliax (judge) decreased severity to Medium and commented:

> Great find, but I think it should be of Medium severity because it requires an external condition, the owner should stop excluding the contract, and also in case that happens, function setIsExcludedAddresses can be used to exclude this address again so the funds are not stuck forever in this case.

[M-16] `WhitelistPeriodManager` : Improper state handling of exclusion additions

Submitted by hickuphh3

WhitelistPeriodManager.sol#L178-L184
WhitelistPeriodManager.sol#L83-L99

The `totalLiquidity` and `totalLiquidityByLp` mappings are not updated when an addres is added to the `isExcludedAddress` mapping. This affects the enforcement of the cap limits and the `getMaxCommunityLpPositon()` function, which implicitly assumes that whitelisted addresses will have 0 liquidity, for addresses with non-zero liquidity at the time of addition to the whitelist.

Proof of Concept

- Assume the following initial conditions:
    - Alice's address `0xA` is the sole USDC liquidity provider
        - `totalLiquidity[USDC] = 500`
        - `totalLiquidity[USDC][0xA] = 500`
    - USDC total cap of `500` , ie. `perTokenTotalCap[USDC] = 500`
- Exclude Alice's address `0xA` : `setIsExcludedAddressStatus([0xA, true])`
    - totalLiquidity mappings are unchanged
- The following deviant behaviour is observed:
    - `getMaxCommunityLpPositon()` returns `500` when it should return `0`
    - All non-excluded addresses are unable to provide liquidity when they should have been able to, as Alice's liquidity should have been excluded.

```
// insert test case in WhitelistPeriodManager.test.ts
describe.only("Test whitelist addition", async () => {
it('produces deviant behaviour if excluding address with existing liquidit
await wlpm.setCaps([token.address], [500], [500]);
await liquidityProviders.connect(owner).addTokenLiquidity(token.address, 5
await wlpm.setIsExcludedAddressStatus([owner.address], [true]);
// 1) returns 500 instead of 0
console.log((await wlpm.getMaxCommunityLpPositon(token.address)).toString
// 2) bob (or other non-excluded addresses) will be unable to add liquidit
await expect(liquidityProviders.connect(bob).addTokenLiquidity(token.addre
});
});
```

Recommended Mitigation Steps

Check that the address to be excluded is not holding any LP token at the time of exclusion.

```
// in setIsExcludedAddressStatus()
for (uint256 i = 0; i < _addresses.length; ++i) {
  if (_status[i]) {
    require(lpToken.balanceOf(_addresses[i]) == 0, 'address has exis
  }
  ...
}
```

ankurdubey521 (Biconomy) confirmed

pauliax (judge) commented:

> I think it is a different issue than M-15, based on the description it deserves a severity of Medium.

[M-17] wrong condition checking in price calculation

Submitted by CertoraInc

LiquidityProviders.sol#L180-L186

The `getTokenPriceInLPShares` function calculates the token price in LP shares, but it check a wrong condition - if supposed to return `BASE_DIVISOR` if the total reserve is zero, not if the total shares minted is zero. This might leads to a case where the price is calculated incorrectly, or a division by zero is happening.

Proof of Concept

This is the wrong function implementation:

```
function getTokenPriceInLPShares(address _baseToken) public view ret
    uint256 supply = totalSharesMinted[_baseToken];
    if (supply > 0) {
        return totalSharesMinted[_baseToken] / totalReserve[_baseTok
    }
    return BASE_DIVISOR;
```

```
    }
```

This function is used in this contract only in the removeLiquidity and claimFee function, so it's called only if funds were already deposited and totalReserve is not zero, but it can be problematic when other contracts will use this function (it's a public view function so it might get called from outside of the contract).

Recommended Mitigation Steps

The correct code should be:

```
function getTokenPriceInLPShares(address _baseToken) public view ret
    uint256 reserve = totalReserve[_baseToken];
    if (reserve > 0) {
        return totalSharesMinted[_baseToken] / totalReserve[_baseTok
    }
    return BASE_DIVISOR;
}
```

ankurdubey521 (Biconomy) confirmed and commented:

> HP-25: C4 Audit Fixes, Dynamic Fee Changes bcnmy/hyphen-contract#42

pauliax (judge) commented:

> Great catch, even though the real impact is not that clear and severe, I will favor a warden and leave it as Medium severity.

Pedroais (warden) commented:

> The warden didn't show any attack path that could leak value. This is a view function that is incorrect as to spec so I think this should be a low.

pauliax (judge) commented:

> Yes, it is a view function but nevertheless, I think it possesses a hypothetical risk path that this function can fail at runtime if the totalSharesMinted is 0. It is somewhere between low and medium categories, I am curious what other certified wardens think about where should this belong.

[M-18] Possible frontrun on deposits on `LiquidityPool`

Submitted by Cantor_Dust, also found by WatchPug

Rewards are given to a user for depositing either ERC20 tokens or their native token into the LiquidityPool. This reward is used to incentivize users to deposit funds into the liquidity pool when the pool is not in an equilibrium state.

For regular users, this liquidity pool state fluctuates based on the frequency and amount of deposits made to the liquidity pool. If a malicious user can control the state of the liquidity poo before a victim deposits tokens into the liquidity pool, they can gain double rewards.

To gain these double rewards, a malicious user can watch the mempool for transactions that will receive a reward when the deposit occurs. When a malicious user sees that victim deposit the malicious user can attach a higher fee to their transaction and initiate a deposit. This will allow the malicious user's transaction to front-run before the victim's transaction.

Once the malicious user's deposit is complete, the liquidity pool state will be in a near equilibrium state. Then, the victim's deposit will occur which causes the liquidity pool state to no longer be in equilibrium.

Finally, the malicious user will make a final deposit gaining yet another reward for bringing the liquidity pool state back to equilibrium.

To sum up, a malicious user can create a sandwich attack where they deposit their own tokens before and after a victim's transaction. This will allow the malicious user to double dip and gain rewards twice due to victim's deposit.

Proof of Concept

Let's look at the depositNative function which is the simpler of the two deposit functions.

The key component in the depositNative function is the getRewardAmount which can be found here. The getRewardAmount calculates how much available vs supplied liquidity exists in the liquidity pool. Here there are no time-weighted checks to calculate the available vs. supplied liquidity. With a lack of checks for time-weight and that there are no frontrun checks against deposits, it's trivial to front-run deposits and control the liquidity of the liquidity such that the reward amount can be double-dipped.

Recommended Mitigation Steps

1. By allowing each deposit to manipulate the liquidity pool state from either a deficient or excessive state, malicious users can double dip on rewards.
2. Alternative approaches to calculating rewards is possible, for example a dutch auction style deposit system where rewards are distributed evenly could reduce an impact of a frontrun attack.
3. A simpler approach is to record liquidity states at specific block timestamps and check against the timestamp for the current block state.

ankurdubey521 (Biconomy) acknowledged

pauliax (judge) commented:

> Great find, mempool lurking monsters could definitely use this opportunity.

[M-19] `sharesToTokenAmount` : Division by zero

Submitted by cmichel, also found by cccz and Certoralnc

LiquidityProviders.sol#L192

The public `sharesToTokenAmount` function does not check if the denominator `totalSharesMinted[_tokenAddress]` is zero.
Neither do the callers of this function. The function will revert.
Calling functions like `getFeeAccumulatedOnNft` and `sharesToTokenAmount` from another contract should never revert.

Recommended Mitigation Steps

Return 0 in case `totalSharesMinted[_tokenAddress]` is zero.

ankurdubey521 (Biconomy) confirmed

pauliax (judge) commented:

> A valid concern of runtime error.

[M-20] Liquidity providers unable to remove liquidity when the pool is in deficit state

Submitted by kyliek

[LiquidityProviders.sol#L388](#)
[LiquidityProviders.sol#L392](#)

LP token holders can not redeem their tokens when the pool is in the deficit state, i.e. `currentLiquidity << providedLiquidity`. This is due to that LP shares are computed based on `providedLiquidity` and the actual available pool balance is based on `currentLiquidity`.

Proof of Concept

When a high valued withdrawal happens in the liquidity pool of the destination chain, the current liquidity will be reduced when the executor calls `sendFundsToUser`
[LiquidityPool.sol#L285](#)

and the pool contract balance will also be reduced by the same amount. The pool reached a deficit state with provided liquidity much bigger than current liquidity.

The LP shares are computed based on the value of `totalReserve` that is roughly equivalent to `totalLiquidity + LpFees`. In a deficit state, `totalReserve` could be much bigger than the available pool balance (up to 90% since max fee is 10%). If the LP token holder wants to redeem his shares,

```
_decreaseCurrentLiquidity(_tokenAddress, _amount);
```

will underflow and revert and

```
_transferFromLiquidityPool(_tokenAddress, _msgSender(), amou
```

will revert because there is not enough balance.

Recommended Mitigation Steps

This is a tricky problem. On one hand, separating `currentLiquidity` from `providedLiquidity` made sure that by bridging tokens over, it will not inflate or deflate the pool. On the other hand, decoupling the two made it hard to compute the actual available liquidity to redeem LP shares. One may need to think through this a bit more.

ankurdubey521 (Biconomy) disagreed with High severity and commented:

> Liquidity Providers will be able to withdraw their funds as long as they're sufficient `currentLiquidity` in the pool, as you mentioned. This will be the case when all pools are balanced, ie the current liquidity is very close to the supplied liquidity.

> By design, hyphen liquidity pools incentivize people to rebalance the pools by providing rewards from the incentive pool, so we believe this should not be that big of an issue in practice.

pauliax (judge) decreased severity to Medium and commented:

> A valid concern, and even though per the sponsor's comment it should not be a problem in practice, a hypothetical path of risk still exists so I would like to leave this as of Medium severity issue.

Low Risk and Non-Critical Issues

For this contest, 39 reports were submitted by wardens detailing low risk and non-critical issues. The report highlighted below by **hickuphh3** received the top score from the judge.

The following wardens also submitted reports: cmichel, 0x1f8b, catchup, rfa, CertoraInc, PPrieditis, IIIIIII, Ruhum, gzeon, minhquanym, bitbopper, 0xngndev, benk10, Dravee, kenta, kyliek, defsec, saian, samruna, berndartmueller, robee, danb, jayjonah8, hubble, WatchPug, 0xwags, Cantor_Dust, TerrierLover, 0xNazgul, csanuragjain, throttle, z3s, ye0lde, XDms, cryptphi, hagrid, 0v3rf10w, *and* 0xDjango.

Codebase Impressions & Summary

Overall, code quality for the Hyphen 2.0 contracts is high. Supporting documentation was adequate in helping to understand the incentive and fee mechanisms for cross-chain transfers.

The contracts in scope have 81.36% statement and 54.91% branch test coverage. Notably, the Liquidity Pool's `permitAndDepositErc20()` and `permitEIP2612AndDepositErc20()` functions that allow users to deposit with signed messages are untested. It will be ideal to write more tests so that better coverage is achieved. Also note that some liquidity farming tests often fail because rewards are continuously accruing, so the actual amount tends to be greater than the expected amount.

[L-01] Conflicting values of `BASE_DIVISOR`

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityPool.sol#L20

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityProviders.sol#L27

Description

`BASE_DIVISOR` is `10_000_000_000` in `LiquidityPool`, but `10**18` in `LiquidityProviders`. This can easily confuse 3rd parties integrating the token bridge.

Recommended Mitigation Steps

Rename either variable. I recommend renaming the instance in `LiquidityPool` to `FEE_DIVISOR`.

[L-02] Sub-optimal calculations in `getAmountToTransfer()` results in wei losses

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityPool.sol#L317-L322

Description

In the scenario where the transfer fee exceeds the equilibrium fee, the excess gets credited to the incentive pool. Disregarding from the fact that a bracket is incorrectly placed causing a massive loss in incentives (raised in separate issue), there are cases where 1 wei is unaccounte for from precision loss in the calculation.

```
    lpFee = (amount * tokenManager.getTokensInfo(tokenAddress).equilibri
    // altered for bracket positioning
    incentivePool[tokenAddress] +=
      (amount * (transferFeePerc - tokenManager.getTokensInfo(tokenAddre
      BASE_DIVISOR;
```

Proof of Concept

- `amount = 337671308498`
- `transferFeePerc = 181480242`
- `equilibriumFee = 10000000` (0.1%)

Calculated amounts are

- `lpFee = 337671308`
- `incentive = 337671308498 * (181480242 - 10000000) / BASE_DIVISOR = 5790395769`

Total fee calculated = `337671308 + 5790395769 = 6128067077`

- `transferFeeAmount = 337671308498 * 181480242 / BASE_DIVISOR = 6128067078`

We therefore see 1 wei being unaccounted for.

Recommended Mitigation Steps

```
uint256 transferFeeAmount = (amount * transferFeePerc) / BASE_DIVISC
uint256 lpFee;
uint256 equilibriumFee = tokenManager.getTokensInfo(tokenAddress).eq
if (transferFeePerc > equilibriumFee) {
  lpFee = amount * equilibriumFee / BASE_DIVISOR;
  incentivePool[tokenAddress] += transferFeeAmount - lpFee;
} else {
...
}
```

[L-03] Unbounded iterations for `getMaxCommunityLpPositon()`

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/WhitelistPeriodManager.sol#L248

Description

The `getMaxCommunityLpPosition()` iterates through the LP token supply to obtain the maximum community LP position obtained. Because the supply of NFT tokens is uncapped, there will come a point where this function runs out of gas.

Proof of Concept

In the worst case, the limit seems to be at about 1250 NFTs where the (N+1)th LP token has more liquidity than the Nth LP token.

```
it.only("Tries to get max iterations possible for getMaxCommunityLpP
  let MAX_LOOPS = 1250;
  // summation formula for 1 to MAX_LOOPS
  let maxCap = MAX_LOOPS * (MAX_LOOPS + 1) / 2;
  await wlpm.setCaps([token.address], [maxCap], [maxCap]);
  for (let i = 1; i <= MAX_LOOPS; i++) {
    console.log(`adding ${i}`);
    // worst case: every iteration in getMaxCommunityLpPositon() ent
    // by giving next tokenId more liquidity
    await liquidityProviders.connect(owner).addTokenLiquidity(token.
  }
  console.log('getting max lp position...');
  // Runs out of gas
  // Error: Transaction reverted and Hardhat couldn't infer the reas
  await wlpm.getMaxCommunityLpPositon(token.address);
});
```

Recommended Mitigation Steps

Have start and end indexes as inputs to cap the number of iterations performed.

[L-04] `addSupportedToken()` allows zero fees to be set, but `changeFee()` doesn't

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/token/TokenManager.sol#L49-L53

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/token/TokenManager.sol#L91-L98

Description

As per the title, the `addSupportedToken()` allows for zero `equilibriumFee` or `maxFee` to b
set, but `changeFee()` doesn't.

Recommended Mitigation Steps

Either include non-zero checks in `addSupportedToken()` or remove them in `changeFee()`.

[L-05] `_sendErc20AndGetSentAmount()` uses recipient instead of sender balance difference

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityFarming.sol#L109-L117

Description

The function name implies that the sent amount should be returned, but it uses the amount received by the recipient instead.

```
    uint256 recepientBalance = _token.balanceOf(_to);
    _token.safeTransfer(_to, _amount);
    return _token.balanceOf(_to) - recepientBalance;
```

If a fee-on-transfer token is the reward token, the amount sent vs received would differ.

Recommended Mitigation Steps

Decide which value is to be returned and logged. Either update the function to be `_sendErc20AndGetReceivedAmount()` or change it to use the contract's balance difference instead.

```
    uint256 senderBalance = _token.balanceOf(address(this));
    _token.safeTransfer(_to, _amount);
    return _token.balanceOf(address(this)) - senderBalance;
```

[L-06] Add constructor initializer in implementation contracts

Description

As per [OpenZeppelin's (OZ) recommendation](), "The guidelines are now to make it impossible for *anyone* to run `initialize` on an implementation contract, by adding an empty constructor with the `initializer` modifier. So the implementation contract gets initialized automatically upon deployment."

Note that this behaviour is also incorporated the [OZ Wizard]() since the UUPS vulnerability discovery: "Additionally, we modified the code generated by the [Wizard]() to include a constructor that automatically initializes the implementation when deployed."

Recommended Mitigation Steps

Add an empty constructor method to the relevant upgradeable contracts.

```
/// @custom:oz-upgrades-unsafe-allow constructor
constructor() initializer {}
```

[L-07] Consider having limit on gas fee charged

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityPool.sol#L330-L336

Description

There is no limit to the gas fee charged. While it is claimed that "there are no incentives for the executors", in reality, executors can be indirectly incentivised by inflating the gas price so that they will be credited a higher fee.

The fee can be as high as the bridged amount - transfer fee, leaving nothing for the user. Whil there is a lot of trust placed on the executor already, it would help to be able to provide a trust less solution by enforcing a cap on the gas fee.

Recommended Mitigation Steps

Limit the maximum gas fee chargeable.

[N-01] Typo errors

- `reightful` → `rightful`
- `Claculate` → `Calculate`
- `sushi` → `reward`

[N-02] Missing underscore for error

Line Reference

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityFarming.sol#L103

Description

The format seems to be 2 underscores after `ERR`, but the line reference above only has 1 underscore: `ERR_REWARD_TOKEN_IS_ZERO`.

Recommended Mitigation Steps

`ERR_REWARD_TOKEN_IS_ZERO -> ERR__REWARD_TOKEN_IS_ZERO`

[N-03] Swap comment order

Line Reference

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityPool.sol#L351

Description

The variable order in the comment do not correspond to that of the implementation. For readability, I recommend that they do.

Recommended Mitigation Steps

```
    uint256 numerator = providedLiquidity * equilibriumFee * maxFee; //
```

[N-04] Deep factor not customisable

Reference

https://biconomy.notion.site/Self-Balancing-Cross-Chain-Liquidity-Pools-c19a725673964d5aaec6b16e5c7ce9a5

Description

The fee calculation formula mentions a deep factor `d`: *Value that decides how much deeper the curve looks.* Readers may have the impression that this may therefore be a customisable parameter in the contract, but in actual factor, is set to a constant value of `1`.

Recommended Mitigation Steps

Users / readers should be made aware that the deep factor has been fixed.

[N-05] Incorrect comment for description of `BASE_DIVISOR`

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityPool.sol#L20

Description

`BASE_DIVISOR` is defined as `10_000_000_000;` with accompanying description `// Basis Points * 100 for better accuracy`.

This isn't accurate as `100% = 10,000` basis points, and `10,000 * 100 = 1_000_000`, not `10_000_000_000`.

Recommended Mitigation Steps

Either update the comment to be

```
uint256 private constant BASE_DIVISOR = 10_000_000_000; // 100 * (Basis Points ^ 2) for better accuracy
```

or the `BASE_DIVISOR` itself to be a different value.

[N-06] Standardize fee denomination

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityPool.sol#L20

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityPool.sol#L350

Description

In relation to L02, there are conflicting definitions of the fee denomination. `BASE_DIVISOR` says that fees are in `Basis points * 100`, while the comment in `getTransferFees()` says they are specified in `basis points * 10`.

Recommended Mitigation Steps

Standardize the fee denomination.

[N-07] Incorrect comment for address to use for withdrawing native token

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityFarming.sol#L175

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityFarming.sol#L186

Description

The comment says to `use 0x00 for Ethereum`, but the implementation uses `NATIVE` instead.

Recommended Mitigation Steps

`use 0x00 for Ethereum` → `use NATIVE for native token`

[N-08] Clarify reserve variable descriptions

Line References

https://github.com/code-423n4/2022-03-biconomy/blob/main/contracts/hyphen/LiquidityProviders.sol#L42-L44

Description

It is unclear what each variable consists of, because there is:

- Supplied liquidity (SL) from liquidity providers
- Available liquidity: SL + net deposits and withdrawals from bridging activity
- Incentive fees
- Gas fees
- LP fees (accumulated equilibrium fees)

Recommended Mitigation Steps

It would be best to explictly state what each variable consists of for clarity.

```
mapping(address => uint256) public totalReserve; // Supplied Liquidi
mapping(address => uint256) public totalLiquidity; // Supplied Liqui
// Available liquidity = SL + net deposits and withdrawals from bric
mapping(address => uint256) public currentLiquidity; // Available Li
```

Gas Optimizations

For this contest, 39 reports were submitted by wardens detailing gas optimizations. The report highlighted below by **Dravee** received the top score from the judge.

The following wardens also submitted reports: lllllll, CertoraInc, TerrierLover, saian, 0xngndev, wuwe1, WatchPug, Jujic, benk10, robee, hickuphh3, kenta, throttle, rfa, bitbopper, z3s, berndartmueller, pedroais, PPrieditis, defsec, gzeon, Cantor_Dust, samruna, Tomio, sirhashalot, antonttc, 0v3rf10w, 0x1f8b, 0xDjango, oyc_109, minhquanym, peritoflores, Kenshin, csanuragjain, 0xwags, Kiep, 0xNazgul, *and* hagrid.

Table of Contents

See original submission.

Foreword

- **Storage-reading optimizations**

> The code can be optimized by minimising the number of SLOADs. SLOADs are expensive (100 gas) compared to MLOADs/MSTOREs (3 gas). In the paragraphs below, please see the `@audit-issue` tags in the pieces of code's comments for more information about SLOADs that could be saved by caching the mentioned **storage** variables in **memory** variables.

- **Unchecking arithmetics operations that can't underflow/overflow**

> Solidity version 0.8+ comes with implicit overflow and underflow checks on unsigned integers. When an overflow or an underflow isn't possible (as an example, when a comparison is made before the arithmetic operation, or the operation doesn't depend on user input), some gas can be saved by using an `unchecked` block: https://docs.soliditylang.org/en/v0.8.10/control-structures.html#checked-or-unchecked-arithmetic

- `@audit` **tags**

> The code is annotated at multiple places with `//@audit` comments to pinpoint the issues. Please, pay attention to them for more details.

File: LPToken.sol

function updateTokenMetadata()

```
File: LPToken.sol
89:     function updateTokenMetadata(uint256 _tokenId, LpTokenMetada
90:         external
91:         onlyHyphenPools
92:         whenNotPaused
93:     {
94:         require(_exists(_tokenId), "ERR__TOKEN_DOES_NOT_EXIST");
95:         tokenMetadata[_tokenId] = _lpTokenMetadata;
96:     }
```

Use `calldata` instead of `memory` for `LpTokenMetadata memory _lpTokenMetadata`

When arguments are read-only on external functions, the data location should be `calldata`. Here, `LpTokenMetadata memory _lpTokenMetadata` should be `LpTokenMetadata calldata _lpTokenMetadata`

function tokenURI()

```
112:     function tokenURI(uint256 tokenId)
...
124:         string memory svgData = svgHelper.getTokenSvg(
125:             tokenId,
126:             tokenMetadata[tokenId].suppliedLiquidity, //@audit
127:             ILiquidityProviders(liquidityProvidersAddress).tota
128:         );
129:
130:         string memory description = svgHelper.getDescription(
131:             tokenMetadata[tokenId].suppliedLiquidity,//@audit t
132:             ILiquidityProviders(liquidityProvidersAddress).tota
133:         );
134:
135:         string memory attributes = svgHelper.getAttributes(
136:             tokenMetadata[tokenId].suppliedLiquidity,//@audit t
137:             ILiquidityProviders(liquidityProvidersAddress).tota
138:         );
```

Cache `tokenMetadata[tokenId].suppliedLiquidity`

Storage readings are expensive. Caching this in a memory variable can save around 2 SLOADs

Cache `ILiquidityProviders(liquidityProvidersAddress).totalReserve(tokenAddress)`

External calls are expensive. Caching this in a memory variable can save around 2 external calls.

function _beforeTokenTransfer()

```
180:     function _beforeTokenTransfer(
181:         address from,
182:         address to,
183:         uint256 tokenId
184:     ) internal virtual override(ERC721EnumerableUpgradeable, ER
185:         super._beforeTokenTransfer(from, to, tokenId);
186:
187:         // Only call whitelist period manager for NFT Transfers
188:         if (from != address(0) && to != address(0)) { //@audit-
189:             whiteListPeriodManager.beforeLiquidityTransfer(
```

```
190:                 from,
191:                 to,
192:                 tokenMetadata[tokenId].token,
193:                 tokenMetadata[tokenId].suppliedLiquidity
194:             );
195:         }
196:     }
```

Short-circuiting can save gas

The condition L188 can be short-circuited to provide a happy path with the following optimization:

```
        if (from == address(0) || to == address(0)) {
          return;
        }

        whiteListPeriodManager.beforeLiquidityTransfer(
            from,
            to,
            tokenMetadata[tokenId].token,
            tokenMetadata[tokenId].suppliedLiquidity
        );
```

This way, the gas from evaluating the second condition can be saved in case of minting (in this scenario, we're expecting more minting than burning, therefore making a happy path for it).

File: TokenManager.sol

Tight Packing `struct TokenInfo`

To save 1 slot, the struct should go from:

```
File: ITokenManager.sol
06:     struct TokenInfo {
07:         uint256 transferOverhead; //@audit 32 bytes
08:         bool supportedToken; //@audit 1 byte
09:         uint256 equilibriumFee; //@audit 32 bytes
10:         uint256 maxFee; //@audit 32 bytes
11:         TokenConfig tokenConfig;//@audit 20 bytes
12:     }
```

to

```
06:     struct TokenInfo {
07:         uint256 transferOverhead; //@audit 32 bytes
08:         uint256 equilibriumFee; //@audit 32 bytes
09:         uint256 maxFee; //@audit 32 bytes
10:         TokenConfig tokenConfig;//@audit 20 bytes
11:         bool supportedToken; //@audit 1 byte
12:     }
```

function changeFee()

```
File: TokenManager.sol
44:     function changeFee(
45:         address tokenAddress,
46:         uint256 _equilibriumFee,
47:         uint256 _maxFee
48:     ) external override onlyOwner whenNotPaused {
49:         require(_equilibriumFee != 0, "Equilibrium Fee cannot be
50:         require(_maxFee != 0, "Max Fee cannot be 0");
51:         tokensInfo[tokenAddress].equilibriumFee = _equilibriumFe
52:         tokensInfo[tokenAddress].maxFee = _maxFee;
53:         emit FeeChanged(tokenAddress, tokensInfo[tokenAddress].e
54:     }
```

Storage optimization

Instead of fetching the storage value multiple times from the array, it's possible to save some gas and help the optimizer by using a storage variable:

```
TokenInfo storage _tokenInfo = tokensInfo[tokenAddress];
_tokenInfo.equilibriumFee = _equilibriumFee;
_tokenInfo.maxFee = _maxFee;
```

Emitting storage value

Storage values are being emitted L53. I suggest using:

```
emit FeeChanged(tokenAddress, _equilibriumFee, _maxFee);
```

function setDepositConfig()

```
File: TokenManager.sol
69:     function setDepositConfig(
70:         uint256[] memory toChainId,//@audit should be calldata
71:         address[] memory tokenAddresses,//@audit should be calld
72:         TokenConfig[] memory tokenConfig//@audit should be calld
73:     ) external onlyOwner {
74:         require(
75:             (toChainId.length == tokenAddresses.length) && (toke
76:             " ERR_ARRAY_LENGTH_MISMATCH"
77:         );
78:         for (uint256 index = 0; index < tokenConfig.length; ++ir
79:             depositConfig[toChainId[index]][tokenAddresses[index
80:             depositConfig[toChainId[index]][tokenAddresses[index
81:         }
82:     }
```

Use calldata instead of memory for uint256[] memory toChainId

Use calldata instead of memory for address[] memory tokenAddresses

Use calldata instead of memory for TokenConfig[] memory tokenConfig

Storage usage optimization

I suggest going from:

```
depositConfig[toChainId[index]][tokenAddresses[index]].min = tok
depositConfig[toChainId[index]][tokenAddresses[index]].max = tok
```

to:

```
TokenConfig storage _sTokenConfig = depositConfig[toChainId[ir
_sTokenConfig.min = tokenConfig[index].min;
_sTokenConfig.max = tokenConfig[index].max;
```

function getTokensInfo()

```
115:     function getTokensInfo(address tokenAddress) public view ov
116:         TokenInfo memory tokenInfo = TokenInfo( //@audit can si
117:             tokensInfo[tokenAddress].transferOverhead,
118:             tokensInfo[tokenAddress].supportedToken,
119:             tokensInfo[tokenAddress].equilibriumFee,
120:             tokensInfo[tokenAddress].maxFee,
121:             transferConfig[tokenAddress]
122:         );
123:         return tokenInfo;
124:     }
```

The variable `tokenInfo` is only used once: inline it

I suggest the following optimization:

```
    function getTokensInfo(address tokenAddress) public view override re
        return TokenInfo(
            tokensInfo[tokenAddress].transferOverhead,
            tokensInfo[tokenAddress].supportedToken,
            tokensInfo[tokenAddress].equilibriumFee,
            tokensInfo[tokenAddress].maxFee,
            transferConfig[tokenAddress]
        );
    }
```

I believe we can go further here, as the copy in memory is done manually here. As `TokenConfig` is already contained inside the `TokenInfo` struct, this should be the best option:

```
    function getTokensInfo(address tokenAddress) public view override re
        return tokensInfo[tokenAddress];
    }
```

File: LiquidityFarming.sol

Tight Packing `struct NFTInfo`

To save 1 slot, I suggest going from:

```
File: LiquidityFarming.sol
```

```
29:     struct NFTInfo {
30:         address payable staker; //@audit-info 20 bytes
31:         uint256 rewardDebt; //@audit-info 32 bytes
32:         uint256 unpaidRewards; //@audit-info 32 bytes
33:         bool isStaked; //@audit-info 1 byte
34:     }
```

to

```
File: LiquidityFarming.sol
29:     struct NFTInfo {
30:         uint256 rewardDebt; //@audit-info 32 bytes
31:         uint256 unpaidRewards; //@audit-info 32 bytes
32:         address payable staker; //@audit-info 20 bytes
33:         bool isStaked; //@audit-info 1 byte
34:     }
```

function _sendErc20AndGetSentAmount()

```
File: LiquidityFarming.sol
109:     function _sendErc20AndGetSentAmount(
110:         IERC20Upgradeable _token,
111:         uint256 _amount,
112:         address _to
113:     ) private returns (uint256) {
114:         uint256 recepientBalance = _token.balanceOf(_to);
115:         _token.safeTransfer(_to, _amount);
116:         return _token.balanceOf(_to) - recepientBalance; //@auc
117:     }
```

Uncheck L116

This line can't underflow due to L114-L115. Therefore, it should be wrapped in an `unchecked` block.

function deposit()

Consider adding a function in `ILPToken` to save 1 external call

Here, if a function is added in `ILPToken` to check both conditions in 1 call, it could save 1 external call:

```
File: LiquidityFarming.sol
199:         require(
200:             lpToken.isApprovedForAll(msgSender, address(this))
201:             "ERR__NOT_APPROVED"
202:         );
```

function withdraw()

Uncheck L240

As `nftIdsStaked[msgSender][index] = nftIdsStaked[msgSender][nftIdsStaked[msgSender].length - 1];` can never underflow due to the require statement above it and the for-loop, it should be wrapped in an `unchecked` block.

Use the existing variable `nftsStakedLength` instead of `nftIdsStaked[msgSender].length`

As no push or pop operations are done yet, I suggest going from:

```
File: LiquidityFarming.sol
231:         uint256 nftsStakedLength = nftIdsStaked[msgSender].leng
...
240:         nftIdsStaked[msgSender][index] = nftIdsStaked[msgSender
```

to

```
File: LiquidityFarming.sol
231:         uint256 nftsStakedLength = nftIdsStaked[msgSender].leng
...
240:         nftIdsStaked[msgSender][index] = nftIdsStaked[msgSender
```

function getUpdatedAccTokenPerShare()

```
File: LiquidityFarming.sol
265:     function getUpdatedAccTokenPerShare(address _baseToken) pub
...
274:             unchecked {
275:                 accumulator +=
276:                     rewardRateLog[_baseToken][i].rewardsPerSecc
```

```
277:                     (counter - max(lastUpdatedTime, rewardRateI
278:                 }
279:                 counter = rewardRateLog[_baseToken][i].timestamp; /
280:                 if (i == 0) {
281:                     break;
282:                 }
283:                 --i;//@audit should be unchecked (see L280-L281)
284:             }
...
```

Storage usage optimization

Cache `rewardRateLog[_baseToken][i].timestamp` in memory

Uncheck L283

Here, it's possible to save a substantial amount of gas with the following optimization (taking into account the 3 titles above):

```
            while (true) {
                if (lastUpdatedTime >= counter) {
                    break;
                }
                RewardsPerSecondEntry storage _reward = rewardRateLog[_b
                uint256 _timestamp = _reward.timestamp; //@audit added c
                unchecked {
                    accumulator +=
                        _reward.rewardsPerSecond * // @audit storage opt
                        (counter - max(lastUpdatedTime, _timestamp)); //

                    counter = _timestamp; //@audit MLOAD
                    if (i == 0) {
                        break;
                    }
                    --i;//@audit now unchecked
                }
            }
```

function max()

A private function used only once should get inlined

As `function max()` is a private function (not inherited) that is only used once in the contract (L277), it should get inlined.

File: LiquidityPool.sol

modifier onlyLiquidityProviders()

`modifier onlyLiquidityProviders()` is used only once and should get inlined

As `modifier onlyLiquidityProviders()` is only used once (on `function transfer()`), it should get inlined.

function depositErc20()

Avoid multiple external calls on `tokenManager.getDepositConfig(toChainId, tokenAddress)`

The code can be optimized from:

```
File: LiquidityPool.sol
156:         require(
157:             tokenManager.getDepositConfig(toChainId, tokenAddre
158:                 tokenManager.getDepositConfig(toChainId, tokenA
159:             "Deposit amount not in Cap limit"
160:         );
```

to

```
156:         ITokenManager.TokenConfig memory _depositConfig = toker
157:         require(
158:             _depositConfig.min <= amount && //@audit MLOAD
159:                 _depositConfig.max >= amount, //@audit MLOAD
160:             "Deposit amount not in Cap limit"
161:         );
```

function getRewardAmount()

```
File: LiquidityPool.sol
175:     function getRewardAmount(uint256 amount, address tokenAddre
```

```
176:         uint256 currentLiquidity = getCurrentLiquidity(tokenAdc
177:         uint256 providedLiquidity = liquidityProviders.getSuppl
178:         if (currentLiquidity < providedLiquidity) {
179:             uint256 liquidityDifference = providedLiquidity - c
```

Uncheck L179

As `providedLiquidity - currentLiquidity` can never underflow due to the if statement above it, it should be wrapped in an `unchecked` block.

function depositNative()

Avoid multiple external calls on `tokenManager.getDepositConfig(toChainId, NATIVE)`

The code can be optimized from:

```
File: LiquidityPool.sol
247:         require(
248:             tokenManager.getDepositConfig(toChainId, NATIVE).mi
249:                 tokenManager.getDepositConfig(toChainId, NATIVE
250:             "Deposit amount not in Cap limit"
251:         );
```

to

```
File: LiquidityPool.sol
247:         ITokenManager.TokenConfig memory _depositConfig = token
248:         require(
249:             _depositConfig.min <= msg.value && //@audit MLOAD
250:                 _depositConfig.max >= msg.value, //@audit MLOAD
251:             "Deposit amount not in Cap limit"
252:         );
```

function sendFundsToUser()

Avoid multiple external calls on `tokenManager.getTransferConfig(tokenAddress)`

The code can be optimized from:

```
File: LiquidityPool.sol
272:         require(
273:             tokenManager.getTransferConfig(tokenAddress).min <=
274:                 tokenManager.getTransferConfig(tokenAddress).ma
275:             "Withdraw amnt not in Cap limits"
276:         );
```

to

```
File: LiquidityPool.sol
272:         ITokenManager.TokenConfig memory _transferConfig = toke
273:         require(
274:             _transferConfig.min <= amount && //@audit MLOAD
275:                 _transferConfig.max >= amount, //@audit MLOAD
276:             "Withdraw amnt not in Cap limits"
277:         );
```

Reorder require statements to save gas on revert

Here, there are two require statements:

```
File: LiquidityPool.sol
272:         require(
273:             tokenManager.getTransferConfig(tokenAddress).min <=
274:                 tokenManager.getTransferConfig(tokenAddress).ma
275:             "Withdraw amnt not in Cap limits"
276:         );
277:         require(receiver != address(0), "Bad receiver address")
```

The second require statement is a simple condition that is a lot less expensive than the first one. In case of revert on the second require statement, all the gas from the first require would be wasted (2 external calls, or 1 after the optimization). I suggest reordering the require statements to put this one first.

function getAmountToTransfer()

```
File: LiquidityPool.sol
308:     function getAmountToTransfer(
...
316:         if (transferFeePerc > tokenManager.getTokensInfo(tokenA
```

```
317:             // Here add some fee to incentive pool also
318:             lpFee = (amount * tokenManager.getTokensInfo(tokenA
319:             incentivePool[tokenAddress] =
320:                 (incentivePool[tokenAddress] +
321:                     (amount * (transferFeePerc - tokenManager.g
322:                 BASE_DIVISOR;
```

Avoid multiple external calls on `tokenManager.getTokensInfo(tokenAddress).equilibriumFee`

`tokenManager.getTokensInfo(tokenAddress).equilibriumFee` should get cached to avoid 2 unnecessary external calls.

Uncheck L321

As `transferFeePerc - tokenManager.getTokensInfo(tokenAddress).equilibriumFee` can never underflow due to the if statement above it L316, it should be wrapped in an `unchecked` block.

function getTransferFee()

```
File: LiquidityPool.sol
342:     function getTransferFee(address tokenAddress, uint256 amoun
...
348:         uint256 equilibriumFee = tokenManager.getTokensInfo(tok
349:         uint256 maxFee = tokenManager.getTokensInfo(tokenAddres
...
```

Avoid multiple external calls on `tokenManager.getTokensInfo(tokenAddress)`

I suggest the following optimization:

```
File: LiquidityPool.sol
348:         ITokenManager.TokenInfo memory _tokenInfo = tokenManage
349:         uint256 equilibriumFee = _tokenInfo.equilibriumFee; //@
350:         uint256 maxFee = _tokenInfo.maxFee; //@audit MLOAD
```

File: LiquidityProviders.sol

Storage

```
27:     uint256 public constant BASE_DIVISOR = 10**18; //@audit gas
```

Use `1e18` instead of `10**18` for constant `BASE_DIVISOR`

Due to how `constant` variables are implemented (constant expressions are expressions, not constants), `10**18` will be more expensive than `1e18`.

modifier onlyValidLpToken()

Consider adding a function in `ILPToken` to save 1 external call

Here, the modifier is quite expensive as it makes 2 external calls:

```
File: LiquidityProviders.sol
53:     modifier onlyValidLpToken(uint256 _tokenId, address _transac
54:         (address token, , ) = lpToken.tokenMetadata(_tokenId);
55:         require(lpToken.exists(_tokenId), "ERR__TOKEN_DOES_NOT_E
56:         require(lpToken.ownerOf(_tokenId) == _transactor, "ERR__
57:         _;
58:     }
```

Consider adding a method in `ILPToken` that both checks that `_tokenId` exists and returns the token's owner.

function _increaseCurrentLiquidity()

```
File: LiquidityProviders.sol
135:     function _increaseCurrentLiquidity(address tokenAddress, ui
136:         currentLiquidity[tokenAddress] += amount; //@audit SLOA
137:         emit CurrentLiquidityChanged(tokenAddress, currentLiqui
138:     }
```

Cache `currentLiquidity[tokenAddress]`

Caching this in a memory variable can save around 2 SLOADs. Here's the full optimization:

```
File: LiquidityProviders.sol
135:     function _increaseCurrentLiquidity(address tokenAddress, ui
136:         uint256 _currentLiquidity = currentLiquidity[tokenAddre
137:         uint256 _increasedLiquidity = _currentLiquidity + amoun
138:         currentLiquidity[tokenAddress] = _increasedLiquidity;
139:         emit CurrentLiquidityChanged(tokenAddress, _currentLiqu
140:     }
```

function _decreaseCurrentLiquidity()

```
File: LiquidityProviders.sol
140:     function _decreaseCurrentLiquidity(address tokenAddress, ui
141:         currentLiquidity[tokenAddress] -= amount; //@audit SLOA
142:         emit CurrentLiquidityChanged(tokenAddress, currentLiqui
143:     }
```

Cache `currentLiquidity[tokenAddress]`

Caching this in a memory variable can save around 2 SLOADs. Here's the full optimization:

```
File: LiquidityProviders.sol
140:     function _decreaseCurrentLiquidity(address tokenAddress, ui
141:         uint256 _currentLiquidity = currentLiquidity[tokenAddre
142:         uint256 _decreasedLiquidity = _currentLiquidity - amoun
143:         currentLiquidity[tokenAddress] = _decreasedLiquidity;
144:         emit CurrentLiquidityChanged(tokenAddress, _currentLiqu
145:     }
```

function getTokenPriceInLPShares()

```
File: LiquidityProviders.sol
180:     function getTokenPriceInLPShares(address _baseToken) public
181:         uint256 supply = totalSharesMinted[_baseToken];
182:         if (supply > 0) {
183:             return totalSharesMinted[_baseToken] / totalReserve
184:         }
185:         return BASE_DIVISOR;
186:     }
```

Use `supply` instead of `totalSharesMinted[_baseToken]`

At line 183, I suggest using `supply` instead of `totalSharesMinted[_baseToken]` . Full code:

```
File: LiquidityProviders.sol
180:     function getTokenPriceInLPShares(address _baseToken) public
181:         uint256 supply = totalSharesMinted[_baseToken];
182:         if (supply > 0) {
183:             return supply / totalReserve[_baseToken];
184:         }
185:         return BASE_DIVISOR;
186:     }
```

function _increaseLiquidity()

```
File: LiquidityProviders.sol
280:     function _increaseLiquidity(uint256 _nftId, uint256 _amount
281:         (address token, uint256 totalSuppliedLiquidity, uint256
282:
283:         require(_amount > 0, "ERR__AMOUNT_IS_0");
284:         whiteListPeriodManager.beforeLiquidityAddition(_msgSend
285:
286:         uint256 mintedSharesAmount;
287:         // Adding liquidity in the pool for the first time
288:         if (totalReserve[token] == 0) { //@audit totalReserve[t
289:             mintedSharesAmount = BASE_DIVISOR * _amount;
290:         } else {
291:             mintedSharesAmount = (_amount * totalSharesMinted[t
292:         }
293:
294:         require(mintedSharesAmount >= BASE_DIVISOR, "ERR__AMOUN
295:
296:         totalLiquidity[token] += _amount;
297:         totalReserve[token] += _amount; //@audit totalReserve[t
298:         totalSharesMinted[token] += mintedSharesAmount; //@audi
```

Cache `totalReserve[token]`

Caching this in memory can save around 2 SLOADs

Cache `totalSharesMinted[token]`

Caching this in memory can save around 1 SLOAD (only after 1st liquidity adding in the pool fo the first time)

File: WhitelistPeriodManager.sol

modifier onlyLpNft()

`modifier onlyLpNft()` is used only once should get inlined

As `modifier onlyLpNft()` is only used once (on `function beforeLiquidityTransfer()`), it should get inlined.

function setIsExcludedAddressStatus()

```
File: WhitelistPeriodManager.sol
178:     function setIsExcludedAddressStatus(address[] memory _addre
179:         require(_addresses.length == _status.length, "ERR__LENG
180:         for (uint256 i = 0; i < _addresses.length; ++i) {
181:             isExcludedAddress[_addresses[i]] = _status[i];
182:             emit ExcludedAddressStatusUpdated(_addresses[i], _s
183:         }
184:     }
```

Use `calldata` instead of `memory` for `address[] memory _addresses`

Use `calldata` instead of `memory` for `bool[] memory _status`

function setCaps()

```
File: WhitelistPeriodManager.sol
219:     function setCaps(
220:         address[] memory _tokens, //@audit should be calldata
221:         uint256[] memory _totalCaps,//@audit should be calldata
222:         uint256[] memory _perTokenWalletCaps//@audit should be
223:     ) external onlyOwner {
224:         require(
225:             _tokens.length == _totalCaps.length && _totalCaps.l
226:             "ERR__LENGTH_MISMACH"
227:         );
228:         for (uint256 i = 0; i < _tokens.length; ++i) {
```

```
229:                 setCap(_tokens[i], _totalCaps[i], _perTokenWalletCa
230:             }
231:         }
```

Use `calldata` instead of `memory` for `address[] memory _tokens`

Use `calldata` instead of `memory` for `uint256[] memory _totalCaps`

Use `calldata` instead of `memory` for `uint256[] memory _perTokenWalletCaps`

function ifEnabled()

```
File: WhitelistPeriodManager.sol
260:     function ifEnabled(bool _cond) private view returns (bool)
261:         return !areWhiteListRestrictionsEnabled || (areWhiteLis
262:     }
```

The condition can be optimized to save a SLOAD

`!areWhiteListRestrictionsEnabled || (areWhiteListRestrictionsEnabled && _cond)` should be changed to `!areWhiteListRestrictionsEnabled || _cond` as the 2nd part of this statement will only evaluate if `areWhiteListRestrictionsEnabled == true`, therefore the explicit check isn't necessary.

General recommendations

Version

Upgrade pragma to at least 0.8.4

Using newer compiler versions and the optimizer give gas optimizations. Also, additional safety checks are available for free.

The advantages here are:

- **Low level inliner** (>= 0.8.2): Cheaper runtime gas (especially relevant when the contract has small functions).
- **Optimizer improvements in packed structs** (>= 0.8.3)

- **Custom errors** (>= 0.8.4): cheaper deployment cost and runtime cost. *Note*: the runtime cost is only relevant when the revert condition is met. In short, replace revert strings by custom errors.

Instances include:

```
hyphen/token/LPToken.sol:2:pragma solidity 0.8.0;
hyphen/token/TokenManager.sol:3:pragma solidity 0.8.0;
hyphen/ExecutorManager.sol:3:pragma solidity 0.8.0;
hyphen/LiquidityFarming.sol:2:pragma solidity 0.8.0;
hyphen/LiquidityPool.sol:3:pragma solidity 0.8.0;
hyphen/LiquidityProviders.sol:2:pragma solidity 0.8.0;
hyphen/WhitelistPeriodManager.sol:2:pragma solidity 0.8.0;
```

Consider upgrading pragma to at least 0.8.4.

Variables

No need to explicitly initialize variables with default values

If a variable is not set/initialized, it is assumed to have the default value (`0` for `uint` , `false` for `bool` , `address(0)` for address...). Explicitly initializing it with its default value is an anti-pattern and wastes gas.

As an example: `for (uint256 i = 0; i < numIterations; ++i) {` should be replaced with `for (uint256 i; i < numIterations; ++i) {`

Instances include:

```
hyphen/token/LPToken.sol:77:          for (uint256 i = 0; i < nftIds.l
hyphen/token/TokenManager.sol:78:          for (uint256 index = 0; inc
hyphen/ExecutorManager.sol:31:        for (uint256 i = 0; i < execut
hyphen/ExecutorManager.sol:47:        for (uint256 i = 0; i < execut
hyphen/LiquidityFarming.sol:233:        for (index = 0; index < nfts
hyphen/LiquidityFarming.sol:266:        uint256 accumulator = 0;
hyphen/WhitelistPeriodManager.sol:180:          for (uint256 i = 0; i
hyphen/WhitelistPeriodManager.sol:228:          for (uint256 i = 0; i
hyphen/WhitelistPeriodManager.sol:247:          uint256 maxLp = 0;
```

I suggest removing explicit initializations for default values.

Pre-increments cost less gas compared to post-increments

Comparisons

`> 0` is less efficient than `!= 0` for unsigned integers (with proof)

`!= 0` costs less gas compared to `> 0` for unsigned integers in `require` statements with the optimizer enabled (6 gas)

Proof: While it may seem that `> 0` is cheaper than `!=`, this is only true without the optimizer enabled and outside a require statement. If you enable the optimizer at 10k AND you're in a `require` statement, this will save gas. You can see this tweet for more proofs: https://twitter.com/gzeon/status/1485428085885640706

I suggest changing `> 0` with `!= 0` here:

```
hyphen/LiquidityProviders.sol:239:        require(_amount > 0, "ERR_
hyphen/LiquidityProviders.sol:283:        require(_amount > 0, "ERR_
hyphen/LiquidityProviders.sol:410:        require(lpFeeAccumulated >
```

Also, please enable the Optimizer.

For-Loops

An array's length should be cached to save gas in for-loops

Reading array length at each iteration of the loop takes 6 gas (3 for mload and 3 to place memory_offset) in the stack.

Caching the array length in the stack saves around 3 gas per iteration.

Here, I suggest storing the array's length in a variable before the for-loop, and use it instead:

```
hyphen/token/LPToken.sol:77:        for (uint256 i = 0; i < nftIds.l
hyphen/token/TokenManager.sol:78:        for (uint256 index = 0; inc
hyphen/ExecutorManager.sol:31:        for (uint256 i = 0; i < execut
hyphen/ExecutorManager.sol:47:        for (uint256 i = 0; i < execut
hyphen/WhitelistPeriodManager.sol:180:        for (uint256 i = 0; i
hyphen/WhitelistPeriodManager.sol:228:        for (uint256 i = 0; i
```

Increments can be unchecked

In Solidity 0.8+, there's a default overflow check on unsigned integers. It's possible to uncheck this in for-loops and save some gas at each iteration, but at the cost of some code readability, as this uncheck cannot be made inline.

ethereum/solidity#10695

Instances include:

```
hyphen/token/LPToken.sol:77:          for (uint256 i = 0; i < nftIds.l
hyphen/token/TokenManager.sol:78:          for (uint256 index = 0; inc
hyphen/ExecutorManager.sol:31:          for (uint256 i = 0; i < execut
hyphen/ExecutorManager.sol:47:          for (uint256 i = 0; i < execut
hyphen/LiquidityFarming.sol:233:          for (index = 0; index < nfts
hyphen/WhitelistPeriodManager.sol:180:          for (uint256 i = 0; i
hyphen/WhitelistPeriodManager.sol:228:          for (uint256 i = 0; i
hyphen/WhitelistPeriodManager.sol:248:          for (uint256 i = 1; i
```

The code would go from:

```
for (uint256 i; i < numIterations; ++i) {
 // ...
}
```

to:

```
for (uint256 i; i < numIterations;) {
 // ...
 unchecked { ++i; }
}
```

The risk of overflow is inexistant for a `uint256` here.

Visibility

Functions that should be external

According to Slither, these functions should be external to save gas:

- ExecutorManager.getExecutorStatus(address) (contracts/hyphen/Exec
- ExecutorManager.getAllExecutors() (contracts/hyphen/ExecutorManag
- HyphenLiquidityFarming.initialize(address,address,ILiquidityProvi
- HyphenLiquidityFarming.setRewardPerSecond(address,uint256) (contr
- HyphenLiquidityFarming.getNftIdsStaked(address) (contracts/hyphen
- HyphenLiquidityFarming.getRewardRatePerSecond(address) (contracts
- LiquidityPool.initialize(address,address,address,address,address)
- LiquidityPool.setTrustedForwarder(address) (contracts/hyphen/Liqu
- LiquidityPool.setLiquidityProviders(address) (contracts/hyphen/Li
- LiquidityPool.getExecutorManager() (contracts/hyphen/LiquidityPoo
- LiquidityProviders.initialize(address,address,address,address) (c
- LiquidityProviders.getTotalReserveByToken(address) (contracts/hyp
- LiquidityProviders.getSuppliedLiquidityByToken(address) (contract
- LiquidityProviders.getTotalLPFeeByToken(address) (contracts/hyphe
- LiquidityProviders.getCurrentLiquidity(address) (contracts/hyphen
- LiquidityProviders.increaseCurrentLiquidity(address,uint256) (con
- LiquidityProviders.decreaseCurrentLiquidity(address,uint256) (con
- LiquidityProviders.getFeeAccumulatedOnNft(uint256) (contracts/hyp
- WhitelistPeriodManager.initialize(address,address,address,address
- LPToken.initialize(string,string,address,address) (contracts/hyph
- LPToken.setSvgHelper(address,ISvgHelper) (contracts/hyphen/token/
- LPToken.getAllNftIdsByUser(address) (contracts/hyphen/token/LPTok
- LPToken.exists(uint256) (contracts/hyphen/token/LPToken.sol#98-10
- TokenManager.getEquilibriumFee(address) (contracts/hyphen/token/T
- TokenManager.getMaxFee(address) (contracts/hyphen/token/TokenMana
- TokenManager.getTokensInfo(address) (contracts/hyphen/token/Token
- TokenManager.getDepositConfig(uint256,address) (contracts/hyphen/
- TokenManager.getTransferConfig(address) (contracts/hyphen/token/T

Errors

Reduce the size of error messages (Long revert Strings)

Shortening revert strings to fit in 32 bytes will decrease deployment time gas and will decrease runtime gas when the revert condition is met.

Revert strings that are longer than 32 bytes require at least one additional mstore, along with additional overhead for computing memory offset, etc.

Revert strings > 32 bytes:

```
hyphen/token/LPToken.sol:70:        require(_whiteListPeriodManager
hyphen/ExecutorManager.sol:17:        require(executorStatus[msg.ser
```

```
hyphen/LiquidityPool.sol:77:        require(_msgSender() == address(
```

I suggest shortening the revert strings to fit in 32 bytes, or that using custom errors as described next.

Use Custom Errors instead of Revert Strings to save Gas

Custom errors from Solidity 0.8.4 are cheaper than revert strings (cheaper deployment cost and runtime cost when the revert condition is met)

Source: https://blog.soliditylang.org/2021/04/21/custom-errors/:

> Starting from Solidity v0.8.4, there is a convenient and gas-efficient way to explain to users why an operation failed through the use of custom errors. Until now, you could already use strings to give more information about failures (e.g., `revert("Insufficient funds.");`), but they are rather expensive, especially when it comes to deploy cost, and it is difficult to use dynamic information in them.

Custom errors are defined using the `error` statement, which can be used inside and outside of contracts (including interfaces and libraries).

Instances include:

```
hyphen/token/LPToken.sol:52:        require(_msgSender() == liquidit
hyphen/token/LPToken.sol:57:        require(_svgHelper != ISvgHelper
hyphen/token/LPToken.sol:58:        require(_tokenAddress != address
hyphen/token/LPToken.sol:64:        require(_liquidityProviders != a
hyphen/token/LPToken.sol:70:        require(_whiteListPeriodManager
hyphen/token/LPToken.sol:94:        require(_exists(_tokenId), "ERR_
hyphen/token/LPToken.sol:120:        require(svgHelpers[tokenAddress
hyphen/token/TokenManager.sol:16:        require(tokenAddress != add
hyphen/token/TokenManager.sol:17:        require(tokensInfo[tokenAdd
hyphen/token/TokenManager.sol:49:        require(_equilibriumFee !=
hyphen/token/TokenManager.sol:50:        require(_maxFee != 0, "Max
hyphen/token/TokenManager.sol:74:        require(
hyphen/token/TokenManager.sol:91:        require(tokenAddress != add
hyphen/token/TokenManager.sol:92:        require(maxCapLimit > minCa
hyphen/token/TokenManager.sol:110:         require(maxCapLimit > minC
hyphen/ExecutorManager.sol:17:      require(executorStatus[msg.ser
hyphen/ExecutorManager.sol:38:      require(executorAddress != add
hyphen/ExecutorManager.sol:39:      require(!executorStatus[execut
hyphen/ExecutorManager.sol:54:      require(executorAddress != add
hyphen/LiquidityFarming.sol:101:      require(rewardTokens[_baseTo
```

```
hyphen/LiquidityFarming.sol:102:        require(_baseToken != addres
hyphen/LiquidityFarming.sol:103:        require(_rewardToken != addr
hyphen/LiquidityFarming.sol:124:        require(nft.isStaked, "ERR__
hyphen/LiquidityFarming.sol:141:                    require(success,
hyphen/LiquidityFarming.sol:146:                    require(success,
hyphen/LiquidityFarming.sol:185:        require(_to != address(0), "
hyphen/LiquidityFarming.sol:188:            require(success, "ERR__N
hyphen/LiquidityFarming.sol:199:        require(
hyphen/LiquidityFarming.sol:207:        require(rewardTokens[baseTok
hyphen/LiquidityFarming.sol:208:        require(rewardRateLog[baseTo
hyphen/LiquidityFarming.sol:211:        require(!nft.isStaked, "ERR_
hyphen/LiquidityFarming.sol:239:        require(index != nftsStakedI
hyphen/LiquidityFarming.sol:259:        require(nftInfo[_nftId].stak
hyphen/LiquidityPool.sol:72:      require(executorManager.getExecu
hyphen/LiquidityPool.sol:77:      require(_msgSender() == address(
hyphen/LiquidityPool.sol:82:      require(tokenAddress != address(
hyphen/LiquidityPool.sol:83:      require(tokenManager.getTokensIn
hyphen/LiquidityPool.sol:94:      require(_executorManagerAddress
hyphen/LiquidityPool.sol:95:      require(_trustedForwarder != add
hyphen/LiquidityPool.sol:96:      require(_liquidityProviders != a
hyphen/LiquidityPool.sol:108:      require(trustedForwarder != add
hyphen/LiquidityPool.sol:114:      require(_liquidityProviders !=
hyphen/LiquidityPool.sol:128:      require(_executorManagerAddress
hyphen/LiquidityPool.sol:156:      require(
hyphen/LiquidityPool.sol:161:      require(receiver != address(0),
hyphen/LiquidityPool.sol:162:      require(amount != 0, "Amount ca
hyphen/LiquidityPool.sol:247:      require(
hyphen/LiquidityPool.sol:252:      require(receiver != address(0),
hyphen/LiquidityPool.sol:253:      require(msg.value != 0, "Amount
hyphen/LiquidityPool.sol:272:      require(
hyphen/LiquidityPool.sol:277:      require(receiver != address(0),
hyphen/LiquidityPool.sol:281:      require(!status, "Already Proce
hyphen/LiquidityPool.sol:288:          require(address(this).balan
hyphen/LiquidityPool.sol:290:          require(success, "Native Tr
hyphen/LiquidityPool.sol:292:          require(IERC20Upgradeable(t
hyphen/LiquidityPool.sol:373:      require(tokenAddress != NATIVE,
hyphen/LiquidityPool.sol:376:      require(_gasFeeAccumulated != 0
hyphen/LiquidityPool.sol:385:      require(_gasFeeAccumulated != 0
hyphen/LiquidityPool.sol:389:      require(success, "Native Transf
hyphen/LiquidityPool.sol:399:      require(receiver != address(0),
hyphen/LiquidityPool.sol:401:          require(address(this).balan
hyphen/LiquidityPool.sol:403:          require(success, "ERR__NATI
hyphen/LiquidityPool.sol:406:          require(baseToken.balanceOf
hyphen/LiquidityProviders.sol:55:        require(lpToken.exists(_tok
hyphen/LiquidityProviders.sol:56:        require(lpToken.ownerOf(_to
hyphen/LiquidityProviders.sol:64:        require(_msgSender() == add
hyphen/LiquidityProviders.sol:69:        require(tokenAddress != add
hyphen/LiquidityProviders.sol:70:        require(_isSupportedToken(t
hyphen/LiquidityProviders.sol:202:        require(lpToken.exists(_nf
```

```
hyphen/LiquidityProviders.sol:239:        require(_amount > 0, "ERR_
hyphen/LiquidityProviders.sol:252:        require(success, "ERR__NAT
hyphen/LiquidityProviders.sol:268:        require(_token != NATIVE,
hyphen/LiquidityProviders.sol:269:        require(
hyphen/LiquidityProviders.sol:283:        require(_amount > 0, "ERR_
hyphen/LiquidityProviders.sol:294:        require(mintedSharesAmount
hyphen/LiquidityProviders.sol:319:        require(_isSupportedToken(
hyphen/LiquidityProviders.sol:320:        require(token != NATIVE, "
hyphen/LiquidityProviders.sol:321:        require(
hyphen/LiquidityProviders.sol:334:        require(_isSupportedToken(
hyphen/LiquidityProviders.sol:335:        require(token == NATIVE, "
hyphen/LiquidityProviders.sol:337:        require(success, "ERR__NAT
hyphen/LiquidityProviders.sol:352:        require(_isSupportedToken(
hyphen/LiquidityProviders.sol:354:        require(_amount != 0, "ERR
hyphen/LiquidityProviders.sol:355:        require(nftSuppliedLiquidi
hyphen/LiquidityProviders.sol:403:        require(_isSupportedToken(
hyphen/LiquidityProviders.sol:410:        require(lpFeeAccumulated >
hyphen/WhitelistPeriodManager.sol:41:           require(_msgSender() ==
hyphen/WhitelistPeriodManager.sol:46:           require(_msgSender() ==
hyphen/WhitelistPeriodManager.sol:51:           require(tokenAddress !=
hyphen/WhitelistPeriodManager.sol:52:           require(_isSupportedTok
hyphen/WhitelistPeriodManager.sol:92:           require(ifEnabled(total
hyphen/WhitelistPeriodManager.sol:93:           require(
hyphen/WhitelistPeriodManager.sol:179:           require(_addresses.len
hyphen/WhitelistPeriodManager.sol:187:           require(totalLiquidity
hyphen/WhitelistPeriodManager.sol:188:           require(_totalCap >= p
hyphen/WhitelistPeriodManager.sol:203:           require(_perTokenWalle
hyphen/WhitelistPeriodManager.sol:224:           require(
```

I suggest replacing revert strings with custom errors.

Disclosures

C4 is an open organization governed by participants in the community.

C4 Contests incentivize the discovery of exploits, vulnerabilities, and bugs in smart contracts. Security researchers are rewarded at an increasing rate for finding higher-risk issues. Contest submissions are judged by a knowledgeable security researcher and solidity developer and disclosed to sponsoring developers. C4 does not conduct formal verification regarding the provided code but instead provides final verification.

C4 does not provide any guarantee or warranty regarding the security of this project. All smar contract software should be used at the sole risk and responsibility of users.