hexens × 

JULY.24

SECURITY REVIEW REPORT FOR GLIF

CONTENTS

- The isValidReceiver modifier in InfinityPoolV2 and LiquidityMineLP can be bypassed
- InfinityPoolV2.setTreasuryFeeRate() will act retroactively if called on pause
- Validation is missing to ensure the agent was liquidated before `writeOff`
- Unused Agent.setFaulty() leads to inaccurate information
- InfinityPoolV2 does not implement ERC20 functions needed to comply with ERC4626
- `withdrawAndHarvest` function use the same amount for both actions
- There is no way to migrate the miner after upgrading the agent
- Unused _withdraw() function in InfinityPoolV2
- Exploitable Rounding Error in Initial Pool Deposits

ABOUT HEXENS

Hexens is a cybersecurity company that strives to elevate the standards of security in Web 3.0, create a safer environment for users, and ensure mass Web 3.0 adoption.

Hexens has multiple top-notch auditing teams specialized in different fields of information security, showing extreme performance in the most challenging and technically complex tasks, including but not limited to: Infrastructure Audits, Zero Knowledge Proofs / Novel Cryptography, DeFi and NFTs. Hexens not only uses widely known methodologies and flows, but focuses on discovering and introducing new ones on a day-to-day basis.

In 2022, our team announced the closure of a $4.2 million seed round led by IOSG Ventures, the leading Web 3.0 venture capital. Other investors include Delta Blockchain Fund, Chapter One, Hash Capital, ImToken Ventures, Tenzor Capital, and angels from Polygon and other blockchain projects.

Since Hexens was founded in 2021, it has had an impressive track record and recognition in the industry: Mudit Gupta - CISO of Polygon Technology - the biggest EVM Ecosystem, joined the company advisory board after completing just a single cooperation iteration. Polygon Technology, 1inch, Lido, Hats Finance, Quickswap, Layerswap, 4K, RociFi, as well as dozens of DeFi protocols and bridges, have already become our customers and taken proactive measures towards protecting their assets.

EXECUTIVE SUMMARY

OVERVIEW

This audit covered a new version of the Pool contracts within the Glif protocol, the foundational DeFi primitive of Filecoin. The protocol allows Filecoin token holders to earn sustainable rewards on their FIL by lending it to a diverse pool of Filecoin Miners. This new version focuses on migrating from InfinityPool to InfinityPoolV2 and from AgentPolice to AgentPoliceV2. Additionally, it introduces a reward mechanism for Liquidity Providers, enabling them to earn rewards by holding iFIL tokens.

Our security assessment involved a comprehensive review of both new and existing smart contracts over a period of four weeks.

During the audit, we identified one critical and one high vulnerabilities. The critical issue was the lack of verification for agents, which could allow an attacker to deploy a "fake" agent contract and drain rewards from the `LiquidityMineSP` contract. The high one involved a scenario where users could potentially lose their tokens when executing the redeem action.

Additionally, we identified five medium-severity issues, along with several minor vulnerabilities and code optimizations.

All the reported issues were either fixed or acknowledged by the development team and subsequently validated by us.

We can confidently state that the overall security and code quality have improved following the completion of our audit.

SCOPE

The analyzed resources are located on:

https://github.com/glif-confidential/pools/releases/tag/v2.0.0-prerelease-1

The issues described in this report were fixed in the following commit:

https://github.com/glif-confidential/pools/commit/45171361a1f3c843762d990ee5d544eac61ae1d7

AUDITING DETAILS

STARTED
08.07.2024

DELIVERED
05.08.2024

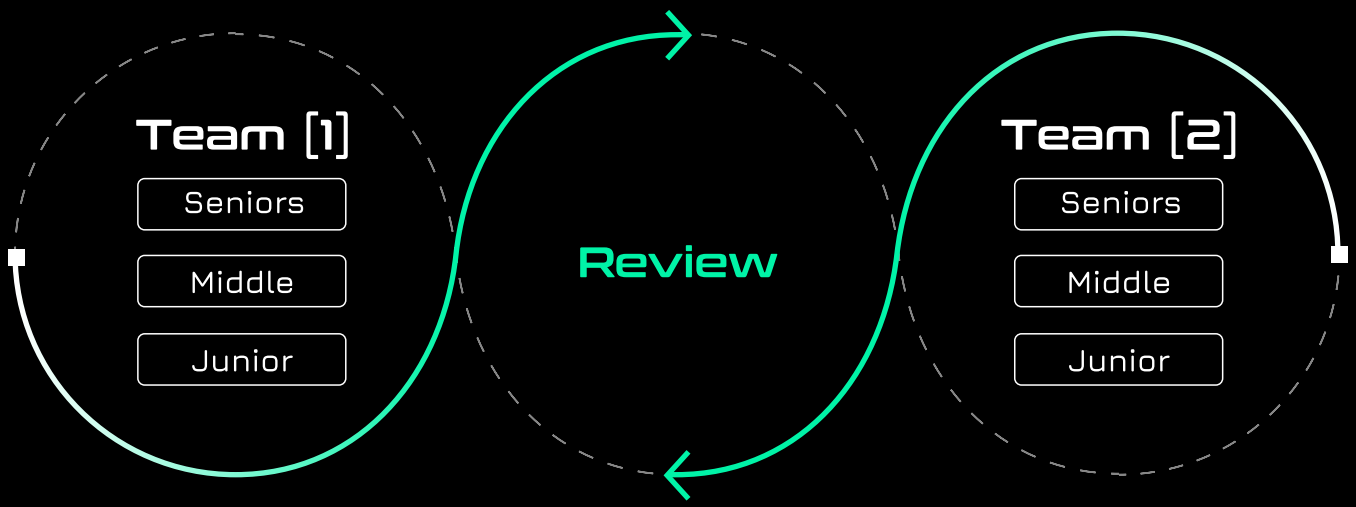
Review Led by

TRUNG DINH

Senior Security Researcher | Hexens

HEXENS METHODOLOGY

Hexens methodology involves 2 teams, including multiple auditors of different seniority, with at least 5 security engineers. This unique cross-checking mechanism helps us provide the best quality in the market.



SEVERITY STRUCTURE

The vulnerability severity is calculated based on two components

- Impact of the vulnerability
- Probability of the vulnerability

Impact	Probability			
	rare	unlikely	likely	very likely
Low/Info	Low/Info	Low/Info	Medium	Medium
Medium	Low/Info	Medium	Medium	High
High	Medium	Medium	High	Critical
Critical	Medium	High	Critical	Critical

SEVERITY CHARACTERISTICS

Smart contract vulnerabilities can range in severity and impact, and it's important to understand their level of severity in order to prioritize their resolution. Here are the different types of severity levels of smart contract vulnerabilities:

Critical

Vulnerabilities with this level of severity can result in significant financial losses or reputational damage. They often allow an attacker to gain complete control of a contract, directly steal or freeze funds from the contract or users, or permanently block the functionality of a protocol. Examples include infinite mints and governance manipulation.

High

Vulnerabilities with this level of severity can result in some financial losses or reputational damage. They often allow an attacker to directly steal yield from the contract or users, or temporarily freeze funds. Examples include inadequate access control integer overflow/underflow, or logic bugs.

Medium

Vulnerabilities with this level of severity can result in some damage to the protocol or users, without profit for the attacker. They often allow an attacker to exploit a contract to cause harm, but the impact may be limited, such as temporarily blocking the functionality of the protocol. Examples include uninitialized storage pointers and failure to check external calls.

Low

Vulnerabilities with this level of severity may not result in financial losses or significant harm. They may, however, impact the usability or reliability of a contract. Examples include slippage and front-running, or minor logic bugs.

Informational

Vulnerabilities with this level of severity are regarding gas optimizations and code style. They often involve issues with documentation, incorrect usage of EIP standards, best practices for saving gas, or the overall design of a contract. Examples include not conforming to ERC20, or disagreement between documentation and code.

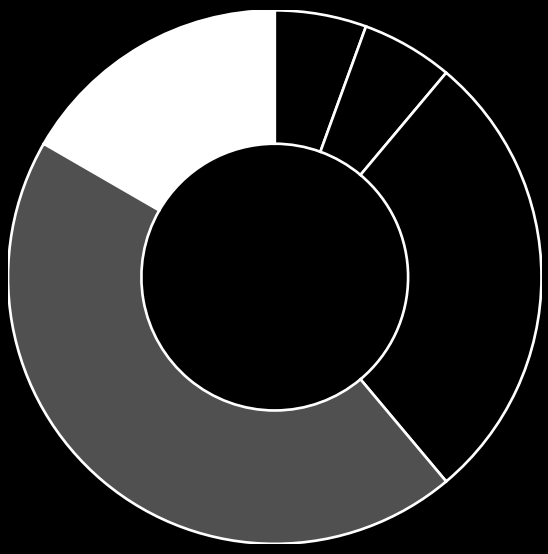
ISSUE SYMBOLIC CODES

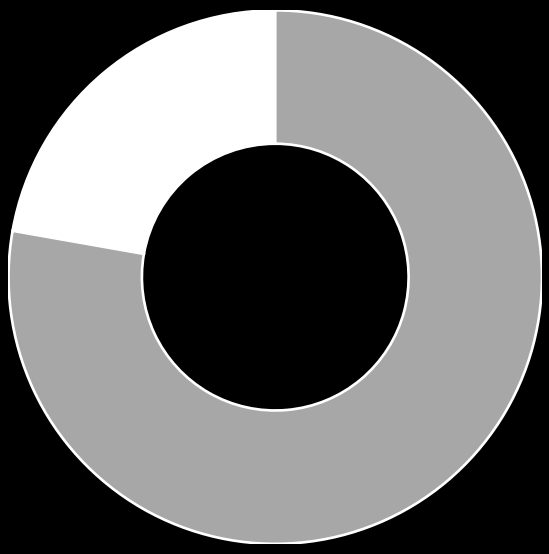
Every issue being identified and validated has its unique symbolic code assigned to the issue at the security research stage. Cause of the vulnerability reporting flow design, some of the rejected issues could be missing.

FINDINGS SUMMARY

Severity	Number of Findings
Critical	1
High	1
Medium	5
Low	8

Total: 18





WEAKNESSES

This section contains the list of discovered weaknesses.

GLIF-20

AUTHORIZATION BYPASS IN LIQUIDITYMINESP ALLOWS ATTACKER TO STEAL UNCLAIMED REWARD TOKENS

SEVERITY: Critical

PATH:

LiquidityMineSP.sol:harvest

REMEDIATION:

The harvest needs to validate that the provided agent is indeed a valid Agent using the AgentFactory:isAgent function.

STATUS: Fixed

DESCRIPTION:

The **harvest** function of the **LiquidityMineSP** can be used by an Agent's owner to withdraw any earned rewards of the Agent. It takes the Agent's address as **agent** parameter. However, this parameter is never validated and it is trusted to be a real Agent.

It makes external calls to this **agent** address for the **id**, **owner** and **defaulted**. But since this is an arbitrary address, these values can be spoofed by the caller.

Since **msg.sender** is only checked against the fetched **owner**, it can be bypassed and consequently the fetched id is then trusted. The corresponding Agent's rewards are used and send to the attacker.

Using this, the attacker can repeat on every agent and drain all of their rewards at once.

ASSET LOSS DUE TO INSUFFICIENT LIQUID ASSETS WHEN USING REDEEM FUNCTION

SEVERITY: High

PATH:

src/Pool/InfinityPoolV2.sol#L863-L869
src/Pool/InfinityPoolV2.sol#L706-L711

REMEDIATION:

Consider reverting the _processExit() function if the assetsToReceive == 0.

STATUS: Fixed

DESCRIPTION:

The function **InfinityPoolV2._redeem()** allows stakers to redeem their shares for assets in two steps:

1. It calculates the amount of assets (FIL) to be transferred to the staker by calling **assets = previewRedeem(shares)**.
2. It processes the exit by burning **shares** iFIL and transferring **assets** FIL to the staker through the **_processExit()** function.

The issue occurs when the calculated **assets** in line 866 of the **previewRedeem()** function exceeds the contract's liquid assets (**getLiquidAssets()**). In this situation, **previewRedeem(shares)** will return **0** assets due to line 868, resulting in the transfer of **0** assets to the staker while still burning **shares** iFIL in the **_processExit()** function.

INCORRECT ROUNDING IN THE PREVIEWREDEEM AND PREVIEWMINT FUNCTIONS

SEVERITY: Medium

PATH:

src/Pool/InfinityPoolV2.sol#L863-L869

src/Pool/InfinityPoolV2.sol#L841-L843

REMEDIATION:

We recommend using rounding down for previewRedeem function and rounding up for previewMint function.

STATUS: Fixed

DESCRIPTION:

The function **InfinityPoolV2::previewRedeem()** attempts to calculate the asset amount received from redeeming a specific share amount. However, it uses rounding up, which will incorrectly return more assets. The **previewRedeem** function should use rounding down, whereas **previewWithdraw** uses rounding up.
For example:
- The total shares are 10 and the total assets are 12
- If a user uses **withdraw** function to withdraw 2 assets, they will need 2 shares.
- If they use **redeem** function, they can redeem only 1 share to obtain 2 assets.

Similarly, **previewMint** function triggers **convertToAssets** and calculates the assets needed to mint incorrectly by rounding down. It should use rounding up instead.

THE MIGRATION CAN BE DOSED

SEVERITY: Medium

PATH:

src/Upgrades/UpgradeToV2.sol#L99
src/Upgrades/PoolSnapshot.sol#L36-L38

REMEDIATION:

We recommend removing the check.

STATUS: Acknowledged

DESCRIPTION:

The **UpgradeToV2.upgrade()** function, responsible for the migration, is prone to DoS because of the internal call to **mustBeEqual()**.

The call contains several checks, one of them is the following:

```
        if (pool1State.totalAssets != pool2State.totalAssets) {
            revert TotalAssetsMismatch(pool1State.totalAssets,
pool2State.totalAssets);
        }
```

The calculations of totalAsset are based on the balance of a pool:

```
    function totalAssets() public view override returns (uint256) {
        // pseudo accounting update to make sure our values are correct
        (uint256 newRentalFeesAccrued, uint256 newTFeesAccrued) =
_computeNewFeesAccrued();
        // using accrual basis accounting, the assets of the pool are:
        // assets currently held by the pool
        // total borrowed from agents
        // total owed rental fees to LPs
        // subtract owed treasury fees
        return asset.balanceOf(address(this)) + totalBorrowed +
_lpRewards.accrue(newRentalFeesAccrued).owed()
            - _treasuryRewards.accrue(newTFeesAccrued).owed();
    }
```

During the migration, the balance of the old pool will be sent to the new one, and the check above ensures it stays the same.

Therefore, a malicious actor can send any number of asset directly to the new pool contract, influencing the accounting and disrupting the check, because old pool balance != old pool balance + 1 wei, for example.

A DEFAULTED AGENT MAY BE UPGRADED TO AVOID LIQUIDATION OR HARVEST LIQUIDITY FEES

SEVERITY: Medium

PATH:

src/Agent/AgentFactory.sol#L51-L78

REMEDIATION:

A defaulted agent shouldn't be allowed to upgrade, so a validation should be added to the upgradeAgent function.

STATUS: Acknowledged

DESCRIPTION:

While upgrading an agent, the **upgradeAgent** function doesn't check if the agent is defaulted, so a defaulted agent can still be upgraded to a new agent. However, the **defaulted** variable of the new agent won't be set to true, allowing it to still harvest rewards from the **LiquidityMineSP** contract. Additionally, the owner of a defaulted agent can use this action to prevent miners liquidation from the **prepareMinerForLiquidation()** function.

THE WRITEOFF() FUNCTION SHOULD NOT BE ALLOWED TO TRIGGER WHEN THE POOL IS PAUSED FOR UPDATING THE INTEREST RATE

SEVERITY: Medium

PATH:

src/Pool/InfinityPoolV2.sol#L291

src/Agent/AgentPoliceV2.sol#L143-L153

REMEDIATION:

Consider adding the whenNotPaused modifier to the function writeOff().

STATUS: Fixed

DESCRIPTION:

When the protocol is updating the interest rate, it will pause operations. Due to transaction gas limits, this rate update process will be divided into multiple transactions. In each transaction, the **epochPaid** for accounts will be recalculated to reflect the new interest rate. However, the STORAGE **_rentalFeesOwedPerEpoch** is updated to the new rate only in the final transaction of the rate update process.

This method leads to incorrect DTL calculations in the **AgentPoliceV2.setAgentDefaultDTL()** function. The DTL is computed based on the current state of the account and the current **getRate()**, which equals **_rentalFeesOwedPerEpoch**. This causes a mismatch during the rate update process. Accounts that have already updated their **epochPaid** will use the new **epochPaid** along with the old interest rate to calculate the **interestOwed**, resulting in an incorrect amount. Consequently resulting in an unexpected liquidation.

INCONSISTENT INTEREST RATE FOR ACCOUNTS DURING MULTI-BLOCK RATE UPDATE PROCESS

SEVERITY: Medium

PATH:

src/Pool/InfinityPoolV2.sol#L1092-L1117

REMEDIATION:

See description.

STATUS: Fixed

DESCRIPTION:

The rate update process involves two functions: startRateUpdate() and **continueRateUpdate()**. The **startRateUpdate()** function is called once, while **continueRateUpdate()** can be called multiple times. This two-step process is necessary because updating a large number of agents in a single transaction could exceed the gas limit.

However, this approach introduces a risk when **startRateUpdate()** and **continueRateUpdate()** are called in different blocks. The problem arises in the **_updateAgentAccounts()** function, where the **interestOwed** for an account is calculated from **epochPaid** to **block.number** using the old interest rate (line 1105). As a result, if **continueRateUpdate()** is triggered in a different block from **startRateUpdate()**, the old interest rate is applied to more blocks for accounts updated by **startRateUpdate()** than for those updated by **continueRateUpdate()**. This creates an unfair situation.

Consider the following example:

1. Assume `_maxAccountsToUpdatePerBatch = 1`, meaning only one account's rate is updated per transaction, and the current `_rentalFeesOwedPerEpoch = 10%`.
2. There are two agents, each with a principal of 100 FIL, who have paid interest up to the current block 10 (`account[1].epochPaid = account[2].epochPaid = 10`).
3. At block 11:
   - The interest owed by the first account is `(11 - 10) * 100 * 10% = 10 FIL`.
   - startRateUpdate() is triggered with `rentalFeesOwedPerEpoch_ = 5%` to update the rate for the first account, and `_updateAgentAccounts() -> _resetEpochsPaid()` updates `account[1].epochPaid = 11 - 10 / 5 = 9`.
4. At block 12:
   - The interest owed by the second account is `(12 - 10) * 100 * 10% = 20 FIL`.
   - `continueRateUpdate()` is triggered to continue the rate update for the second account, and `_updateAgentAccounts() -> _resetEpochsPaid()` updates `account[2].epochPaid = 12 - 20 / 5 = 8`.
   - The rate update process is finished, and `_rentalFeesOwedPerEpoch` is now set to 5%.
5. At block 12, the interest owed by the two accounts is:
   - `interestOwed(account[1], 5%) = (12 - 9) * 100 * 5% = 15 FIL`.
   - `interestOwed(account[2], 5%) = (12 - 8) * 100 * 5% = 20 FIL`.

Despite having the same principal and interest owed before the rate update, the second account owes more interest than the first after the update, creating an unfair scenario for the later-updated account.

Add a new variable to the **RateUpdate** struct to store the **block.timestamp** when **startRateUpdate()** is triggered. This variable can then be used in **continueRateUpdate()** to calculate the **interestOwed** by the accounts.

```
struct RateUpdate {
    uint256 totalAccountsAtUpdate;
    uint256 totalAccountsClosed;
    uint256 newRate;
    bool inProcess;
+    uint rateUpdateStartBlock;
}
```

THE WITHDRAW EVENT WILL BE DUPLICATED IN SEVERAL FUNCTIONS

SEVERITY: Low

PATH:

src/Pool/InfinityPoolV2.sol#L706-L711

REMEDIATION:

We recommend removing the event from the internal function.

STATUS: Fixed

DESCRIPTION:

The following external functions will emit two **Withdraw** events:

- **redeem(uint256 shares, address receiver, address owner)**
- **redeem(uint256 shares, address receiver, address owner, uint256)**

Because the internal **_redeem()** and the underlying **_processExit()** both contain it.

The same issue applies to the internal **_withdraw()**, but it's unused.

LIQUIDITYMINELP._COMPUTEACCREWARDS() TAKES INTO ACCOUNT TOKENS WITHOUT AN OPEN POSITION

SEVERITY: Low

PATH:

src/Token/LiquidityMineLP.sol#L198

REMEDIATION:

We recommend using an internal storage variable to track the total available funds and adding a function to rescue the additional tokens.

STATUS: Fixed

DESCRIPTION:

The internal **_computeAccRewards()** function in **LiquidityMineLP**, responsible for updating the accrued rewards, uses **lockToken.balanceOf(address(this))** to get the total amount of staked tokens.

In case some amount of **lockToken** was sent to the contract directly without using the **deposit()** function, the accounting will still accrue rewards to those tokens, decreasing the rewards for normal depositors. Since the rewards without a position cannot be withdrawn, they will be burned on shutdown.

For example:

1. Alice sends 1 **lockToken** directly to the contract.
2. Bob deposits 1 **lockToken**.
3. Bob waits for 1 week until the mining is over.
4. Bob harvests the rewards. He only gets half of the reward pool, the other half will be locked on the contract and burned later on.

THE ISVALIDRECEIVER MODIFIER IN INFINITYPOOLV2 AND LIQUIDITYMINELP CAN BE BYPASSED

SEVERITY: Low

PATH:

src/Pool/InfinityPoolV2.sol#L122-L125

src/Token/LiquidityMineLP.sol#L62-L65

REMEDIATION:

We recommend normalizing the argument before the checks.

STATUS: Acknowledged

DESCRIPTION:

The **isValidReceiver** modifier does not normalize the **receiver** argument, so it's possible to pass a Filecoin ID instead of an address to bypass the **receiver == address(this)** check.

INFINITYPOOLV2.SETTREASURYFEERATE() WILL ACT RETROACTIVELY IF CALLED ON PAUSE

SEVERITY: Low

PATH:

src/Pool/InfinityPoolV2.sol#L1005-L1008

REMEDIATION:

We recommend adding the whenNotPaused modifier to the function.

STATUS: Fixed

DESCRIPTION:

The **setTreasuryFeeRate()** function in **InfinityPoolV2** is the setter for the **treasuryFeeRate** variable.

Under normal circumstances, it'll perform **updateAccounting()** before to accrue all the fees using the old rate.

But if called during a pause, it'll skip the update and set the variable directly, postponing the calculations to after the pause. When the pause is over, the next call to **updateAccounting()** will unfairly accrue all the fees with the new rate.

VALIDATION IS MISSING TO ENSURE THE AGENT WAS LIQUIDATED BEFORE `WRITEOFF`

SEVERITY: Low

PATH:

src/Agent/AgentPoliceV2.sol#L171-L204

REMEDIATION:

A validation should be added to check the default state of the agent during distributeLiquidatedFunds.

STATUS: Fixed

DESCRIPTION:

During the liquidation of an agent, the **AgentPoliceV2::setAgentDefaultDTL()** function checks the DTL ratio of the agent and sets the storage variable **defaulted** of the agent to true if the DTL is lower than the threshold.

After that, the **AgentPoliceV2::distributeLiquidatedFunds()** function is intended to be called, which triggers the **InfinityPoolV2::writeOff()** function to cover the liquidated debt. This function will set the **account.defaulted** state of the agent's account to true, making the account unable to repay.

```
function writeOff(uint256 agentID, uint256 recoveredFunds) external {
    // only the agent police can call this function
    _onlyAgentPolice();

    updateAccounting();

    Account memory account = _getAccount(agentID);

    if (account.defaulted) revert AlreadyDefaulted();
    // set the account to defaulted
    account.defaulted = true;
```

However, before calling **writeOff**, there is no validation to check that the agent was liquidated (the **defaulted** state of the agent needs to be true). This leads to risks of centralization or mistakes from the owner of **AgentPoliceV2**, as **distributeLiquidatedFunds** can be called and cause lost funds for a healthy agent.

UNUSED AGENT.SETFAULTY() LEADS TO INACCURATE INFORMATION

SEVERITY: Low

PATH:

src/Agent/Agent.sol#L237-L242

REMEDIATION:

We recommend removing the variable and upgrading the Agent, if it's not used by the oracle.

STATUS: Acknowledged

DESCRIPTION:

The **faultySectorStartEpoch** storage variable will always equal 0 because **AgentPoliceV2** does not use its setter, the **setFaulty()** function.

The variable may then be unnecessarily reset in **setRecovered()**.

INFINITYPOOLV2 DOES NOT IMPLEMENT ERC20 FUNCTIONS NEEDED TO COMPLY WITH ERC4626

SEVERITY: Low

PATH:

src/Pool/InfinityPoolV2.sol

REMEDIATION:

We recommend the functions' addition or removal of the comments.

You can use the implementation by OpenZeppelin as a reference: ERC 20 - OpenZeppelin Docs

STATUS: Acknowledged

DESCRIPTION:

There are multiple mentions of the InfinityPoolV2 being an ERC4626 vault, but it does not implement any ERC20 functions.

As you can see in the EIP proposal:

All EIP-4626 tokenized Vaults MUST implement EIP-20 to represent shares.

This means there should be functions available such as **transfer()**, **balanceOf()**, **name()**, etc., in order to conform with the standard.

`WITHDRAWANDHARVEST` FUNCTION USE THE SAME AMOUNT FOR BOTH ACTIONS

SEVERITY: Low

PATH:

src/Token/LiquidityMineLP.sol#L159-L167

REMEDIATION:

We recommend adding another parameter for the harvest amount.

STATUS: Fixed

DESCRIPTION:

In the LiquidityMineLP contract, the **withdrawAndHarvest** function is used to withdraw LP tokens and harvest rewards for users in one transaction. However, it uses the same variable to represent both the LP token amount to withdraw and the reward token amount to harvest, even though they are amounts of different tokens. This leads to users being unable to choose specific amounts for withdrawal and harvesting by this function.

THERE IS NO WAY TO MIGRATE THE MINER AFTER UPGRADING THE AGENT

SEVERITY: Informational

PATH:

src/Agent/Agent.sol#L206-L221

REMEDIATION:

The new agent should have the ability to call the miner, or you can move the responsibility for miner migration to the AgentFactory.

STATUS: Acknowledged

DESCRIPTION:

Regarding the Agent contract, the process for upgrading the agent is as follows: the owner calls **AgentFactory::upgradeAgent()**, which then calls **Agent::decommissionAgent()** and sets up **newAgent**. Afterward, the miner of the old agent should be migrated to the new agent using the **migrateMiner** function in the old Agent contract.
However, this function requires **msg.sender** to be **newAgent**, and an Agent contract does not have the ability to call this function.

```
if (newAgent != msg.sender) revert Unauthorized();
```

To enable migration, the workflow for miner migration should resemble the upgrading process, where the owner can call the AgentFactory, and the AgentFactory will in turn call **Agent::migrateMiner()** to change the miner's new agent.

UNUSED _WITHDRAW() FUNCTION IN INFINITYPOOLV2

SEVERITY: Informational

PATH:

src/Pool/InfinityPoolV2.sol#L792-L797

REMEDIATION:

We recommend modifying the _withdraw() function to accept bool shouldConvert and replacing the duplicate code with this function in:

- withdraw(uint256 assets, address receiver, address owner)
- withdraw(uint256 assets, address receiver, address owner, uint256)
- withdrawF(uint256 assets, address receiver, address owner)
- withdrawF(uint256 assets, address receiver, address owner, uint256)

STATUS: Fixed

DESCRIPTION:

The internal **_withdraw()** function is not used anywhere in the system. The external functions duplicate the internal's functionality.

EXPLOITABLE ROUNDING ERROR IN INITIAL POOL DEPOSITS

SEVERITY: Informational

PATH:

src/Pool/InfinityPoolV2.sol#L226-L236

REMEDIATION:

We recommend that the contract implements a mechanism similar to UniswapV2, which transfers the first 1000 LP tokens to address(0).

STATUS: Acknowledged

DESCRIPTION:

The function **InfinityPoolV2.deposit()** calculates the number of shares a user will receive using the function **previewDeposit() -> convertToShares()**, which utilizes the round-down formula:

```
shares = assets.mulDivDown(supply, totalAssets())
```

where:

- **assets**: the amount of **asset** tokens the user will use to mint shares
- **totalAssets()**: the balance of **asset** tokens in the liquidity pool contract
- **supply**: the current total supply of **liquidStakingToken**

The issue arises when the **totalAssets()** function is calculated directly using a call to **asset.balanceOf(address(this))** in line 234. This flaw can be exploited by the first depositor, as outlined in the following attack strategy:

1. The attacker becomes the first depositor and deposits 1 **asset** token into the liquidity pool, receiving 1 share.
    - **asset.balanceOf(liquidityPool) = 1**
    - **liquidStakingToken.totalSupply() = 1**
2. Alice, an innocent user, intends to deposit **2*10^18** baseTokens to mint shares.
3. The attacker learns of Alice's transaction and front-runs her by transferring **1e18** directly to the liquidity pool.
    - **asset.balanceOf(liquidityPool) = 1e18 + 1**
    - **liquidStakingToken.totalSupply() = 1**
4. When Alice's transaction is executed, she receives 1 share following this calculation:
    - **assets = 2*10^18**
    - **shares = 2*10^18 * 1 / (1e18 + 1) = 1**
    - **asset.balanceOf(liquidityPool) = 3e18 + 1**
    - **liquidStakingToken.totalSupply() = 2**
5. When Alice calls **redeem()** to withdraw all of her **asset** tokens, she only receives:
    - **Line 866: 1 * (3e18 + 1) / 2 = 1.5e18 + 1**

    instantly losing **0.5e18 - 1** tokens, which are profited by the attacker.