fuzzland

WELabs Audit Report

Signature: c6e96edc4bd186790254b4de8f7993bebf74f0b19babf17434f63384773790ef

WELabs Audit Report

Scope

Project Name	WELabs
Repo	https://github.com/WELabsPerpCodeBase/NewPerp
Commit	`4b3f1e62b44aa97bb2f76102b21513f4e2ab905d`
Fix Commit	`14be5d9f0c00ece8e4e4838d4c82b0b3a0ecccde`
Language	Solidity
Scope	contracts/liquidity/LiquidityManager.sol contracts/position/DecreasePositionHandler.sol contracts/position/IncreasePositionHandler.sol contracts/position/PositionLib.sol contracts/repo/Dao.sol contracts/vault/VaultHandler.sol contracts/vault/VaultLib.sol

Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.
- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry and ItyFuzz, to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Vulnerability Severity

We divide severity into three distinct levels: high, medium, low. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	0	0
Medium Severity Issues	2	2
Low Severity Issues	1	1
Info Severity Issues	5	5

Findings

[Med] LP token transfer bypasses liquidity removal cooldown

In the `LiquidityManager` contract, the cooldown mechanism tracks the last liquidity addition time using the `lastAddedAt[_poolToken][_account]` mapping. When users attempt to remove liquidity, the contract checks if the cooldown period has passed:

```
// contracts/liquidity/LiquidityManager.sol
if (lastAddedAt[_poolToken][_account] + cooldownDuration > block.timestamp) {
    revert IsCooldown();
}
```

However, since LP tokens are freely transferable, users can bypass this cooldown restriction by transferring their LP tokens to a new address, as the new address has no record in the `lastAddedAt` mapping.

Recommendation:

Consider associating the cooldown tracking mechanism with the LP tokens themselves rather than user addresses. This could be implemented through a locking period in the LP token contract or by tokenizing LP positions as NFTs to track the last addition time for each token.

Status: Acknowledged.

[Med] Handlers can front-run other handlers' position opening transactions

In the `IncreasePositionHandler` contract, when a user initiates a position opening transaction through a handler, other handlers can observe this pending transaction and front-run it. Specifically, when user funds have been transferred to the contract but the transaction is not yet executed, a malicious handler can preemptively execute a position opening operation, using these funds for another account:

```
// contracts/position/IncreasePositionHandler.sol
function increasePosition(PositionLib.IncreasePositionCache memory cache)
external override nonReentrant onlyHandler {
    // Any handler can use funds in the contract
    uint256 collateralDelta = VaultLib.transferIn(dataBase, address(this),
cache.collateralToken);
    VaultLib.transferOut(dataBase, address(this), cache.collateralToken,
collateralDelta, cache.poolToken);
    // ...
}
```

This design allows malicious handlers to front-run other handlers' transactions through MEV (Miner Extractable Value) mechanisms.

Recommendation:

Record the handler source during transferIn.

Status: Acknowledged.

[Low] Division by zero risk in `vaultLib.getNextGlobalShortAveragePrice()`

The `getNextGlobalShortAveragePrice()` function in VaultLib contains a potential division by zero vulnerability. When calculating the next global short average price, the function
performs division using a divisor that could potentially be zero under specific market conditions.

The issue occurs in the following calculation:

```
// contracts/vault/VaultLib.sol
uint256 divisor = hasProfit ? nextSize - delta : nextSize + delta;
return _nextPrice * nextSize / divisor;
```

When `hasProfit` is true `(meaning averagePrice > _nextPrice)`, the divisor becomes `nextSize - delta`. If `delta` equals `nextSize`, this will result in a division by zero error, causing the transaction to revert.

Recommendation:

Add a check to prevent division by zero:

```
function getNextGlobalShortAveragePrice(DataBase _dataBase, address
_poolToken, address _indexToken, uint256 _nextPrice, uint256 _sizeDelta)
internal view returns (uint256) {
    // ... existing code ...

    uint256 nextSize = size + _sizeDelta;
    uint256 divisor = hasProfit ? nextSize - delta : nextSize + delta;

    require(divisor > 0, "Division by zero");

    return _nextPrice * nextSize / divisor;
}
```

Status: Fixed

[Info] Insufficient input validation

The `addLiquidity()` function in `LiquidityManager.sol`
accepts address parameters (`_token`)
without performing explicit validation checks on these inputs.

```
function addLiquidity(
    address _poolToken,
    address _token,
    uint256 _amount,
    uint256 _minUsdg,
    uint256 _minLp
) external nonReentrant returns (uint256) {
    return _addLiquidity(msg.sender, msg.sender, _poolToken, _token, _amount,
_minUsdg, _minLp);
}
```

Recommendation:

it's recommended to add validation for the `_token` parameter to ensure system robustness.

Status: Acknowledged

[Info] Invalid bounds check

In the following code, `tokenAmount` is defined as a uint256 type:

```
// contracts/vault/VaultHandler.sol
uint256 tokenAmount = VaultLib.transferIn(dataBase, address(this), _token);
if (tokenAmount <= 0) {
    revert Errs.NoTokenAmountIn(_token);
}
...
if (usdgAmount <= 0) {
    revert Errs.InvalidUSDGAmount();
}
```

Since uint256 is an unsigned integer and can never be less than zero, `tokenAmount < 0` in the condition `tokenAmount <= 0` is invalid. This check is redundant, makes the code less readable, and wastes gas.

Recommendation:

Remove the invalid check and modify the code to only check if `tokenAmount == 0`

Status: Fixed

[Info] External code dependency risks (LiquidityLib/PriceLib/RoleBase/DataBase)

The contract performs several key operations through `VaultLib` and `dataBase` and `PriceLib`, including fund transfer, fund transfer, price calculation and fee collection:

Since `VaultLib` and `dataBase` and `PriceLib` are not within the scope of the current audit, the security, integrity, and behavior of their internal implementations are unknown, but these dependent components directly affect:

- The transfer of user assets in and out (transferIn and transferOut).
- Fund-related calculations (price, fee adjustment, quantity accuracy).
- USDG minting and fund pool parameter updates.

Such designs that rely on external libraries or modules may have the following risks:

The code of the external library or module may contain vulnerabilities or malicious behavior, resulting in the theft of funds or the destruction of system functions.

Changes in the logic of the external library or module may cause the contract behavior to be inconsistent with expectations.

Status: Acknowledged

[Info] To prevent silent failure, use `safeTransfer` instead of `transfer`

The `LiquidityManager::_removeLiquidity` function uses `transfer` instead of `safeTransfer` when transferring USDG. This may cause the transfer to fail silently due to the lack of return value check. It is recommended to use `safeTransfer` instead of `transfer`.

```
// contracts/liquidity/LiquidityManager.sol
function _removeLiquidity(address _account, address _poolToken, address _toke
nOut, uint256 _lpAmount, uint256 _minOut, address _receiver) private returns
(uint256) {
//......
IERC20(usdg).transfer(vaultHandler, usdgAmount);
//......
}
```

Recommendation:
use `safeTransfer` instead of `transfer`

Status: Fixed

[Info] Missing 0 address check

When calling `setGov` to add important privileged roles, check whether the authorization address is 0, otherwise the `setGov` function may become permanently invalid.It is also recommended to add 0 address check in `DecreasePositionHandler`, `IncreasePositionHandler`, and `VaultHandler` contracts.

```
// contracts/liquidity/LiquidityManager.sol
function setGov(address _gov) external onlyGov {
    gov = _gov;
    emit UpdateGov(_gov);
}
```

Recommendation:

Add 0 address check.

Status: Fixed