



WEconomy Perpetual Audit Report

WEconomy Perpetual Audit Report

Overview

Scope

Disclaimer

Auditing Process

Vulnerability Severity

Findings

[High] Logic error in `_distribute` function

[High] Users can set their address as `_executionFeeReceiver` to receive `_...`

[Med] Inconsistent execution fee handling

[Med] Incomplete array length validation

[Med] Missing check on `msg.value`

[Low] Use of unsafe ERC20 transfer functions

[Low] Lack of path length validation

[Low] Unsafe token handling and validation

[Low] Lack of iteration limit in loop

[Info] Duplicate permission checks

[Info] Not emit an event

[Info] Missing zero address check


[Info] Unreachable code

[Info] Setting `_timestamp` is meaningless

Overview

From Oct 10, 2024, to Oct 20, 2024, the WEconomy team engaged Fuzzland to conduct a thorough security audit of their Perpetual project. The primary objective was to identify and mitigate potential security vulnerabilities, risks, and coding issues to enhance the project's robustness and reliability. Fuzzland conducted this assessment over 20 person-days, involving 2 engineers who reviewed the code over a span of 10 days. Employing a multifaceted approach that included static analysis, fuzz testing, formal verification, and manual code review, the Fuzzland team identified 14 issues across different severity levels and categories.

Scope

Project Name	WEconomy Perpetual CodeBase
Repository Link	 PerpForAudit
Commit	1479d9042560b268e55d6895c3c80ca311583b64
Fix Commit	85e666c6dcc0d7f7bef8992dc1fd0e8dd26fe382
Language	Solidity
Scope	contracts/**/*.*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.
- **Informational Severity Issues** represent informational findings that do not directly impact the system's security, functionality, or performance. These findings are typically observations or recommendations for potential improvements or optimizations. Addressing info severity issues can enhance the system's robustness and efficiency but is not necessary for the system's immediate operation or security. These issues can be considered for future development or enhancement plans.

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	2	2
Medium Severity Issues	3	3
Low Severity Issues	4	4
Info Severity Issues	5	5

Findings

[High] Logic error in `_distribute` function

In the `_distribute` function of the `ReferralDistributor` contract, when `_typeId` is `traderDiscountsTypeId`, `claimedDiscounts` is erroneously used instead of `claimableDiscounts`.

```
        claimableRebates[recipient][token] = claimableRebates[recipient]
[token].add(amount);
    } else if (_typeId == traderDiscountsTypeId) {
        claimableDiscounts[recipient][token] = claimedDiscounts[recipient]
[token].add(amount); // @audit
    }
}
```

Recommendation:

Change `claimedDiscounts` to `claimableDiscounts`:

```
claimableDiscounts[recipient][token] = claimableDiscounts[recipient]
[token].add(amount);
```

Status: Resolved

[High] Users can set their address as `_executionFeeReceiver` to receive `_executionFee`

The `executeIncreasePosition` and `executeDecreasePosition` functions are `public` and can be called by anyone. The function does not have any check on `_executionFeeReceiver`, which means that any user can set their own address to `_executionFeeReceiver` to collect `_executionFee` that do not belong to them.

```
//PositionRouterSide.sol
function executeIncreasePosition(bytes32 _key, address payable
_executionFeeReceiver) public nonReentrant returns (bool) {
    //.....
    _transferOutETHWithGasLimitFallbackToWeth(request.executionFee,
_executionFeeReceiver);
    //.....
}
```

Recommendation:

Restrict permissions on the `executeIncreasePosition` function.

Status: Resolved

[Med] Inconsistent execution fee handling

In the `SwapRouter` contract, the `createSwapTokensToETH` function allows order creation without sending ETH, but the `cancelSwap` function attempts to refund an execution fee when cancelling orders, potentially leading to unexpected behavior. The `createSwapTokensToETH` function is marked as `payable` but doesn't require users to send ETH:

```
function createSwapTokensToETH(address[] memory _path, uint256 _amountIn,
uint256 _minOut, address _receiver, uint256 _executionFee) external payable
nonReentrant returns (bytes32) {
    require(_path[_path.length - 1] == weth, "invalid _path");
    require(_executionFee >= minExecutionFee, "fee");
    require(_amountIn > 0, "invalid amount");

    _transferInETH();
    IERC20(_path[0]).safeTransferFrom(_sender(), address(this),
    _amountIn);

    return _createSwap(_sender(), _path, _amountIn, _minOut, _receiver,
    _executionFee);
}
```

Users can create orders without sending any ETH as an execution fee. However, the `cancelSwap` function attempts to refund an `executionFee` regardless of whether ETH was actually received when creating the order:

```
_transferOutETHWithGasLimitFallbackToWeth(request.executionFee,
    _executionFeeReceiver);
```

If no ETH was received when creating the order, the cancellation operation might fail due to the contract not having ETH to refund.

Recommendation:

Modify the `createSwapTokensToETH` function to require users to send the correct amount of ETH as an execution fee:

```

function createSwapTokensToETH(address[] memory _path, uint256 _amountIn,
uint256 _minOut, address _receiver, uint256 _executionFee) external payable
nonReentrant returns (bytes32) {
    require(_path[_path.length - 1] == weth, "invalid _path");
    require(msg.value == _executionFee, "incorrect execution fee");
    require(_executionFee >= minExecutionFee, "fee too low");
    require(_amountIn > 0, "invalid amount");

    IERC20(_path[0]).safeTransferFrom(_sender(), address(this),
_amountIn);

    return _createSwap(_sender(), _path, _amountIn, _minOut, _receiver,
_executionFee);
}

```

Status: Resolved

[Med] Incomplete array length validation

In the `_distribute` function of the `ReferralDistributor` contract, there's a lack of consistency check between `_tokens.length` and `_amounts[i].length`.

The function only checks if `_recipients.length` equals `_amounts.length`, but doesn't verify that each `_amounts[i]` length matches the `_tokens` length. This could lead to array out-of-bounds errors when accessing `_amounts[i][j]`.

```

function traderDistribute(
    address[] memory _recipients,
    address[] memory _tokens,
    uint256[][] memory _amounts
) external {
    _onlyHandler();
    _distribute(traderDiscountsTypeId, _recipients, _tokens, _amounts);
}

function _distribute(
    uint256 _typeId,
    address[] memory _recipients,
    address[] memory _tokens,
    uint256[][] memory _amounts
) internal {
    require(_recipients.length == _amounts.length, "Invalid input:
recipients and amounts length mismatch");
    require(_tokens.length > 0, "No tokens provided");

    for (uint256 i = 0; i < _recipients.length; i++) {
        address recipient = _recipients[i];
        uint256[] memory _accountAmounts = new uint256[](_tokens.length);

        for (uint256 j = 0; j < _tokens.length; j++) {
            address token = _tokens[j];
            uint256 amount = _amounts[i][j];
            _accountAmounts[j] = amount;
            if (amount > 0) {
                reserveAmounts[token] =
reserveAmounts[token].add(amount);
                if (_typeId == affiliateRewardsTypeId) {
                    claimableRebates[recipient][token] =
claimableRebates[recipient][token].add(amount);
                } else if (_typeId == traderDiscountsTypeId) {
                    claimableDiscounts[recipient][token] =
claimedDiscounts[recipient][token].add(amount);
                }
            }
        }
        emit DistributeRewards(_typeId, recipient, _tokens,
_accountAmounts);
    }
}

```

Recommendation:

Add an additional check before the loop starts:

```
function _distribute(  
  uint256 _typeId,  
  address[] memory _recipients,  
  address[] memory _tokens,  
  uint256[][] memory _amounts  
) internal {  
  require(_recipients.length == _amounts.length, "Invalid input: recipients  
and amounts length mismatch");  
  require(_tokens.length > 0, "No tokens provided");  
  for (uint256 i = 0; i < _amounts.length; i++) {  
    require(_amounts[i].length == _tokens.length, "Invalid input: tokens  
and amounts length mismatch");  
  }  
  // ... rest of the code remains unchanged  
}
```

Status: Resolved

[Med] Missing check on `msg.value`

According to the semantics of several other swap functions, the `_transferInETH` function is paying `_executionFee`. The `createSwapTokensToETH` function does not add `require(msg.value == _executionFee, "val");` so users may pay less or no `_executionFee`

```
function createSwapTokensToETH(address[] memory _path, uint256 _amountIn,
uint256 _minOut, address _receiver, uint256 _executionFee) external payable
nonReentrant returns (bytes32) {
    require(_path[_path.length - 1] == weth, "invalid _path");
    require(_executionFee >= minExecutionFee, "fee");
    require(_amountIn > 0, "invalid amount");

    _transferInETH();
    IERC20(_path[0]).safeTransferFrom(_sender(), address(this),
    _amountIn);

    return _createSwap(_sender(), _path, _amountIn, _minOut, _receiver,
    _executionFee);
}
```

Recommendation:

Add `require(msg.value == _executionFee, "val");`

```
function createSwapTokensToETH(address[] memory _path, uint256 _amountIn,
uint256 _minOut, address _receiver, uint256 _executionFee) external
nonReentrant returns (bytes32) {
    require(msg.value == _executionFee, "val");
    // ... rest of the code remains unchanged
}
```

Status: Resolved

[Low] Use of unsafe ERC20 transfer functions

In multiple contracts, including ReferralDistributor, BaseLiquidityManager, and SwapRouter, unsafe transfer and transferFrom functions are used for ERC20 token transfers.

Standard transfer and transferFrom functions may return false instead of reverting in some token contracts, or not return anything when failing. This can lead to silent failures, preventing the calling contract from properly handling transfer failures.

```
ReferralDistributor.sol:
214         if (_amounts[i] > 0) {
215:             IERC20(_tokens[i]).transferFrom(msg.sender,
address(this), _amounts[i]);
216         }
BaseLiquidityManager.sol:
99         _weth.deposit{ value: _amountOut }();
100:         _weth.transfer(address(_receiver), _amountOut);
101     }

ReferralDistributor.sol:
203         require(token.balanceOf(address(this)) >= _amount,
"insufficient inventory");
204:         require(token.transfer(_account, _amount), "transfer
failure");
205

223         uint256 balance = IERC20(_token).balanceOf(address(this));
224:         require(IERC20(_token).transfer(_receiver, balance), "withdraw
failed");
225     }

228         _onlyGov();
229:         require(IERC20(_token).transfer(_receiver, _amount), "withdraw
failed");
230     }

SwapRouter.sol:
370         _weth.deposit{ value: _amountOut }();
371:         _weth.transfer(address(_receiver), _amountOut);
372     }
```

Recommendation:

Replace all transfer and transferFrom calls with safeTransfer and safeTransferFrom functions from OpenZeppelin's SafeERC20 library

Status: Resolved

[Low] Lack of path length validation

In the `SwapRouter` contract, the `_createSwap` function lacks validation for the length of the `_path` array. This may lead to unexpected behavior or errors in the subsequent `executeSwap` function. The `createSwap`, `createSwapETHToTokens`, and `createSwapTokensToETH` functions all call the internal `_createSwap` function without checking the length of the `_path` array. In the `executeSwap` function, the `_swap` function only handles paths of length 2 or 3, with other lengths causing the transaction to revert.

```
function createSwap(address[] memory _path, uint256 _amountIn, uint256
_minOut, address _receiver, uint256 _executionFee) external payable
nonReentrant returns (bytes32) {
    require(_executionFee >= minExecutionFee, "fee");
    require(msg.value == _executionFee, "val");
    require(_amountIn > 0, "invalid amount");

    _transferInETH();
    IERC20(_path[0]).safeTransferFrom(_sender(), address(this),
_amountIn);

    return _createSwap(_sender(), _path, _amountIn, _minOut, _receiver,
_executionFee);
}
```

Recommendation:

Add validation for the `_path` length in the `_createSwap` function to ensure it is either 2 or 3. For example:

```
function _createSwap(address _account, address[] memory _path, uint256
_amountIn, uint256 _minOut, address _receiver, uint256 _executionFee)
internal returns (bytes32) {
    require(_path.length == 2 || _path.length == 3, "Invalid path length");
    // ... rest of the code remains unchanged
}
```

Status: Resolved

[Low] Unsafe token handling and validation

In the `LiquidityRouter` contract, the `createWithdrawal` function lacks validation for the `_token` address, while the `executeWithdrawal` function lacks sufficient safety checks when handling non-ETH tokens. These two issues combined can lead to severe security vulnerabilities.

The `createWithdrawal` function allows users to specify any address as `_token` without validity checks:

```
function createWithdrawal(address _token, uint256 _lpAmount, uint256
_minOut, address _receiver, uint256 _executionFee) external payable
nonReentrant returns (bytes32) {
    require(_executionFee >= minExecutionFee, "fee");
    require(msg.value == _executionFee, "val");
    require(_lpAmount > 0, "invalid lpAmount");

    _transferInETH();

    return _createWithdrawal(msg.sender, _token, _lpAmount, _minOut,
_receiver, _executionFee);
}
```

In the `executeWithdrawal` function, the handling of non-ETH tokens lacks adequate safety measures:

```
amountOut = IWlpManager(lpManager).removeLiquidityForAccount(account,
request.token, request.lpAmount, request.minOut, request.receiver);
```

The `removeLiquidityForAccount` function is not within the audit scope and may contain unknown risks.

If `request.token` is a maliciously constructed contract, it might trigger abnormal transfer behavior, returning an incorrect `amountOut` value, and the contract does not verify the actual amount of tokens transferred.

Recommendation:

Implement token address validation in the `createWithdrawal` function.

Status: Resolved

[Low] Lack of iteration limit in loop

The `executeSwaps` function in the `SwapRouter` contract does not limit the number of iterations in a single loop execution. This could lead to the function consuming excessive gas and failing due to exceeding the block gas limit.

The function processes all swap requests between `swapRequestKeysStart` and `_endIndex` in a while loop. If the difference between `_endIndex` and `swapRequestKeysStart` is large, the loop may execute a large number of iterations. Each iteration involves complex operations (executing or cancelling a swap), which consume significant gas. If the total gas consumption exceeds the block gas limit, the transaction will fail.

```
function executeSwaps(uint256 _endIndex, address payable
_executionFeeReceiver) public override onlySwapKeeper {

    uint256 index = swapRequestKeysStart;
    uint256 length = swapRequestKeys.length;

    if (index >= length) { return; }

    if (_endIndex > length) {
        _endIndex = length;
    }
}
```

Recommendation:

1. Introduce a maximum iteration limit.
2. Alternatively, consider implementing a gas check mechanism to exit the loop if remaining gas falls below a threshold.

Status: Resolved

[Info] Duplicate permission checks

There are duplicate permission check logics in the `cancelDeposit` and `cancelWithdrawal` functions, as well as in the `_validateExecutionOrCancellation` function they call.

Both `cancelDeposit` and `cancelWithdrawal` functions contain this check:

```
require(msg.sender == address(this) || isLiquidityKeeper[msg.sender],
"forbidden");
```

These functions then call `_validateCancellation`, which in turn calls `_validateExecutionOrCancellation`, containing a similar check:

```
bool isKeeperCall = msg.sender == address(this) ||
isLiquidityKeeper[msg.sender];
if (!isKeeperCall) {
    revert("forbidden: not keeper");
}
```

This duplication not only leads to code redundancy but could potentially introduce inconsistencies in future code maintenance.

Recommendation:

Remove the initial checks in `cancelDeposit` and `cancelWithdrawal`, relying entirely on the check in `_validateExecutionOrCancellation`.

Status: Resolved

[Info] Not emit an event

Several functions in the code base do not emit relevant events to log their execution.

```
function setGov(address _gov) external {
    _onlyGov();
    gov = _gov;
}

function setHandler(address _account, bool _isActive) external {
    _onlyGov();
    isHandler[_account] = _isActive;
}

function setInPrivateClaimingMode(bool _inPrivateClaimingMode) external {
    _onlyGov();
    inPrivateClaimingMode = _inPrivateClaimingMode;
}

function setRewardTokens(address[] memory _rewardTokens) external {
    _onlyGov();
    rewardTokens = _rewardTokens;
}
```

Recommendation:

Consider defining and emitting events whenever sensitive changes occur.

Status: Acknowledged

[Info] Missing zero address check

In the `constructor` function, the initialization address lacks a zero address check.

```
    constructor(  
        address _weth,  
        address _vault,  
        address _usdg,  
        uint256 _minExecutionFee  
    ) public {  
        weth = _weth;  
        vault = _vault;  
        usdg = _usdg;  
        minExecutionFee = _minExecutionFee;  
        admin = msg.sender;  
    }
```

Recommendation:

It is recommended to add 0 address check.

Status: Resolved

[Info] Unreachable code

`if (_amount > 0)` is repeated in the `createDeposit` function

```
//LiquidityRouter.sol
function createDeposit(address _token, uint256 _amount, uint256 _minUsdg,
uint256 _minLp, uint256 _executionFee) external payable nonReentrant returns
(bytes32) {
    require(_executionFee >= minExecutionFee, "fee");
    require(msg.value == _executionFee, "val");
    require(_amount > 0, "invalid amount");

    _transferInETH();
    if (_amount > 0) {
        IRouter(router).pluginTransfer(_token, msg.sender, address(this),
_amount);
    }

    return _createDeposit(msg.sender, _token, _amount, _minUsdg, _minLp,
_executionFee, false);
}
```

`isKeeperCall` has only two results: false: revert or true: (`return _blockNumber.add(minBlockDelayKeeper) <= block.number;`). All subsequent codes from `require(msg.sender == _account,` are invalid.

```
function _validateExecutionOrCancellation(uint256 _blockNumber, uint256
_blockTime, address _account) internal view returns (bool) {
    bool isKeeperCall = msg.sender == address(this) ||
isLiquidityKeeper[msg.sender];

    if (!isKeeperCall) {
        revert("forbidden: not keeper");
    }

    if (isKeeperCall) {
        return _blockNumber.add(minBlockDelayKeeper) <= block.number;
    }

    require(msg.sender == _account, "forbidden: account error");

    require(_blockTime.add(minTimeDelayPublic) <= block.timestamp,
"delay");

    return true;
}
```


Recommendation:

Delete these invalid codes.

Status: Resolved

[Info] Setting `_timestamp` is meaningless

The only thing the `_setPrices` function does with `_timestamp` is check if it is greater than 0, which doesn't make a lot of sense.

```
//LiquidityRouter.sol
//MultiPriceFeed.sol
//SwapRouter.sol
function _setPrices(int192[] memory _answers, uint256 _timestamp) internal {
    require(_answers.length <= priceFeeds.length, "MultiPriceFeed:
invalid price lengths");
    require(_timestamp > 0, "MultiPriceFeed: invalid timestamp");

    for (uint256 i = 0; i < _answers.length; i++) {
        int192 price = _answers[i];
        if (price > 0) {
            int192[] memory prices = new int192[](1);
            prices[0] = price;

            IPriceFeedV2(priceFeeds[i]).setLatestAnswer(prices);
        }
    }
}
```

Recommendation:

Limit `_timestamp` based on the price update frequency to avoid expired prices, or delete `_timestamp` directly if it is not needed.

Status: Resolved