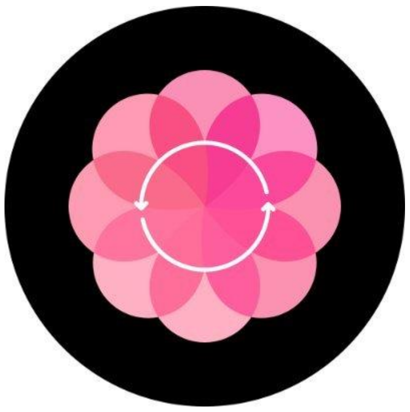
SECURITY AUDIT OF

PERPETUAL EXCHANGE CONTRACT



Public Report

Oct 11, 2023

Verichains Lab

info@verichains.io

https://www.verichains.io

Driving Technology > Forward

ABBREVIATIONS

Name	Description
Ethereum	An open source platform based on blockchain technology to create and distribute smart contracts and decentralized applications.
Ether (ETH)	A cryptocurrency whose blockchain is generated by the Ethereum platform. Ether is used for payment of transactions and computing services in the Ethereum network.
Smart contract	A computer protocol intended to digitally facilitate, verify or enforce the negotiation or performance of a contract.
Solidity	A contract-oriented, high-level language for implementing smart contracts for the Ethereum platform.
Solc	A compiler for Solidity.
ERC20	ERC20 (BEP20 in Binance Smart Chain or *x*RP20 in other chains) tokens are blockchain-based assets that have value and can be sent and received. The primary difference with the primary coin is that instead of running on their own blockchain, ERC20 tokens are issued on a network that supports smart contracts such as Ethereum or Binance Smart Chain.

EXECUTIVE SUMMARY

This Security Audit Report was prepared by Verichains Lab on Oct 11, 2023. We would like to thank the Roseon for trusting Verichains Lab in auditing smart contracts. Delivering high-quality audits is always our top priority.

This audit focused on identifying security flaws in code and the design of the Perpetual Exchange Contract. The scope of the audit is limited to the source code files provided to Verichains. Verichains Lab completed the assessment using manual, static, and dynamic analysis techniques.

During the audit process, the audit team had identified some vulnerable issues in the contract code.

TABLE OF CONTENTS

1. MANAGEMENT SUMMARY

1.1. About Perpetual Exchange Contract

The Perpetual Exchange - Trading Platform leverages cutting-edge market-making technology. Liquidity Providers (LP) can earn fees through market making, swap fees, and leverage trading, offering a lucrative opportunity for traders and LPs alike.

1.2. Audit scope

This audit focused on identifying security flaws in code and the design of the Perpetual Exchange Contract.

It was conducted on commit `8065261d5b1a51a40fc9ddb719377d6d2398eb51` from git repository *https://github.com/roseonx/perpetual-exchange-contracts*.

1.3. Audit methodology

Our security audit process for smart contract includes two steps:

- Smart contract codes are scanned/tested for commonly known and more specific vulnerabilities using public and RK87, our in-house smart contract security analysis tool.
- Manual audit of the codes for security issues. The contracts are manually analyzed to look for any potential problems.

Following is the list of commonly known vulnerabilities that were considered during the audit of the smart contract:

- Integer Overflow and Underflow
- Timestamp Dependence
- Race Conditions
- Transaction-Ordering Dependence
- DoS with (Unexpected) revert
- DoS with Block Gas Limit
- Gas Usage, Gas Limit and Loops
- Redundant fallback function
- Unsafe type Inference
- Reentrancy
- Explicit visibility of functions state variables (external, internal, private and public)
- Logic Flaws

For vulnerabilities, we categorize the findings into categories as listed in table below, depending on their severity level:

SEVERITY LEVEL	DESCRIPTION
CRITICAL	A vulnerability that can disrupt the contract functioning; creates a critical risk to the contract; required to be fixed immediately.
HIGH	A vulnerability that could affect the desired outcome of executing the contract with high impact; needs to be fixed with high priority.
MEDIUM	A vulnerability that could affect the desired outcome of executing the contract with medium impact in a specific scenario; needs to be fixed.
LOW	An issue that does not have a significant impact, can be considered as less important.

Table 1. Severity levels

1.4. Disclaimer

Please note that security auditing cannot uncover all existing vulnerabilities, and even an audit in which no vulnerabilities are found is not a guarantee for a 100% secure smart contract. However, auditing allows discovering vulnerabilities that were unobserved, overlooked during development and areas where additional security measures are necessary.

2. AUDIT RESULT

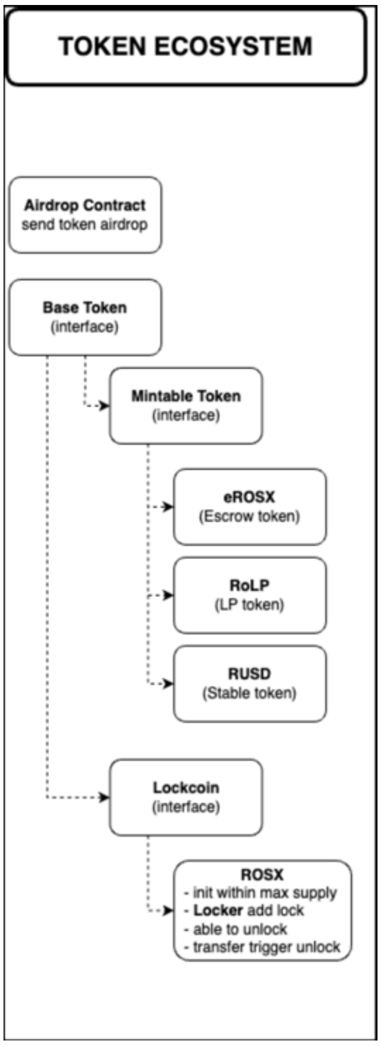
2.1. Overview

The Perpetual Exchange Contract was written in `Solidity` language, with the required version to be `^0.8.12`. The source code was written based on OpenZeppelin's library.

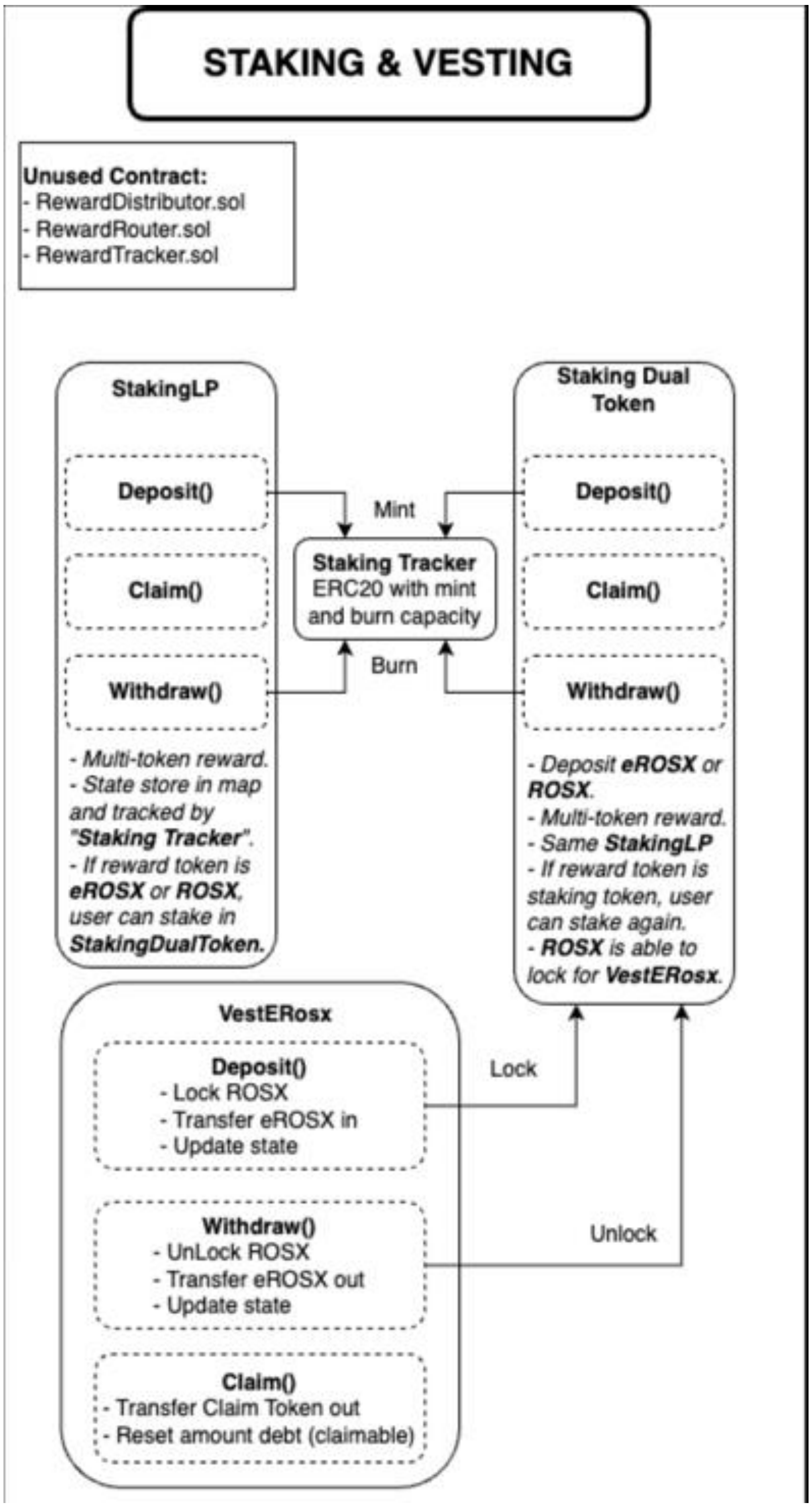
2.1.1. Tokens

The project consists of four types of tokens:

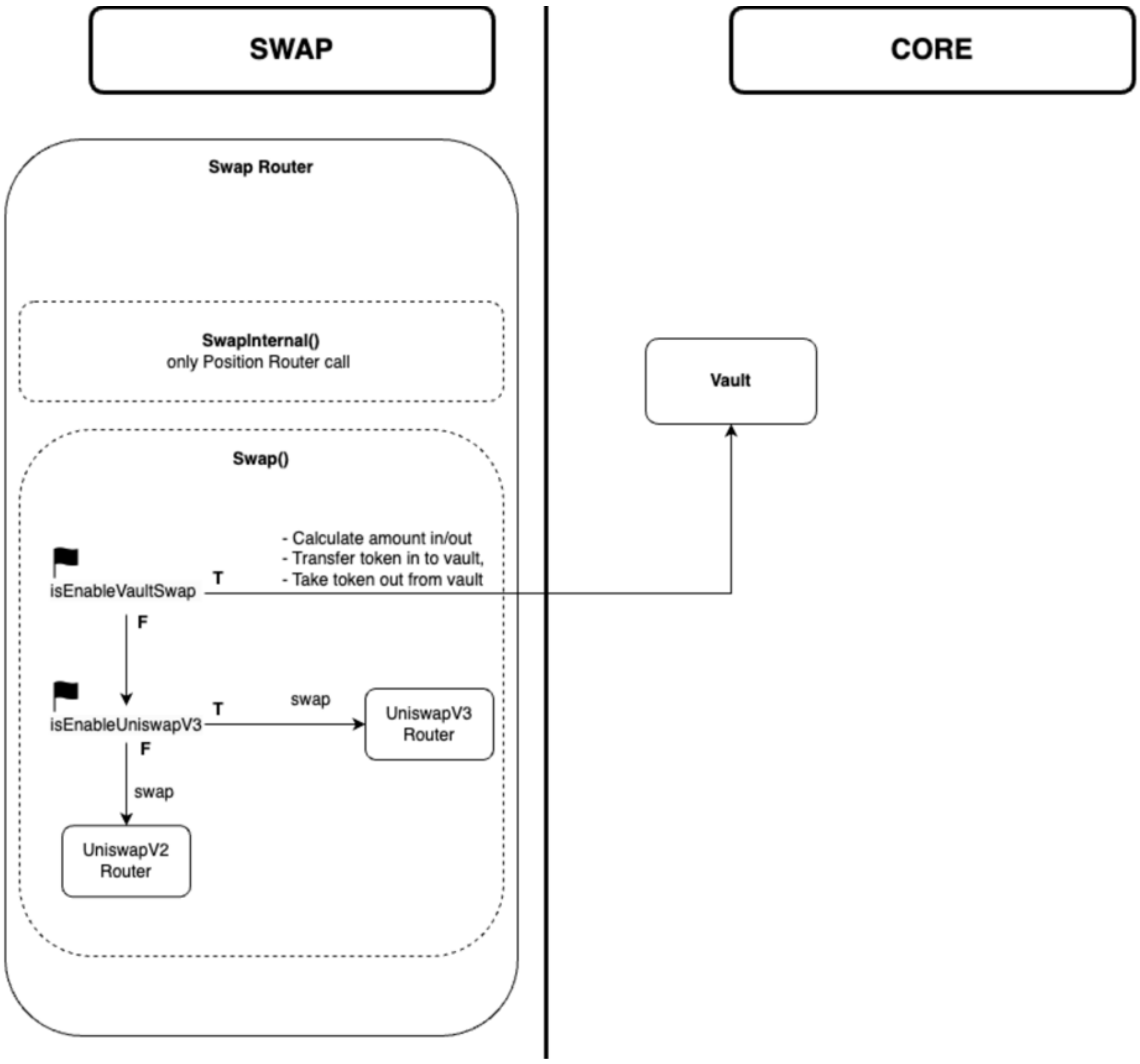
- eROX
- RoLP
- RUSD
- ROSX



2.1.2. Staking


STAKING & VESTING
Unused Contract:
- RewardDistributor.sol
- RewardRouter.sol
- RewardTracker.sol
StakingLP
Deposit()
Claim()
Withdraw()
- Multi-token reward.
- State store in map and tracked by "Staking Tracker".
- If reward token is eROSX or ROSX, user can stake in StakingDualToken.
Mint
Burn
Staking Tracker
ERC20 with mint and burn capacity
Staking Dual Token
Deposit()
Claim()
Withdraw()
- Deposit eROSX or ROSX.
- Multi-token reward.
- Same StakingLP
- If reward token is staking token, user can stake again.
- ROSX is able to lock for VestERosx.
VestERosx
Deposit()
- Lock ROSX
- Transfer eROSX in
- Update state
Lock
Withdraw()
- UnLock ROSX
- Transfer eROSX out
- Update state
Unlock
Claim()
- Transfer Claim Token out
- Reset amount debt (claimable)

2.1.3. Swapping



2.1.4. Core

- **PositionRouter.sol**: The main contract allow users to open/close/add position.
- **SettingsManager.sol**: The contract store the global config.
- **Vault.sol**: The vault store trading/collateral tokens.

2.2. Findings

Severity	Name	Status
CRITICAL	Always reverts when adding a trailing stop if the input path requires a swap	FIXED
CRITICAL	Abuse transfer LP token to bypass cooldown duration check	FIXED
CRITICAL	Decrease pool amount must be in USD unit	FIXED
CRITICAL	Users will lose their assets if the swap fails in opening a new position	FIXED
CRITICAL	Arbitrary swap for any account without permission	FIXED
CRITICAL	Must update token balance after transfer in Vault	FIXED
CRITICAL	The vault swap logic cannot execute because of permission restricted	FIXED
HIGH	Use `tx.origin` for authentication is vulnerable to phishing	FIXED
HIGH	The executor's fund will be drained if there are too many "take profit/stop lost" orders placed	FIXED
HIGH	Get first index of path at index 0 instead of index 1	FIXED
MEDIUM	`_isFastExecute` is always true in `triggerPosition()` function	FIXED
MEDIUM	No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision	FIXED
MEDIUM	Possible reentrancy in all external function	FIXED
MEDIUM	Incorrect position data validation	FIXED

Table 2. List Findings

2.2.1. Always reverts when adding a trailing stop if the input path requires a swap CRITICAL

Positions:

- PositionRouterV2.sol#L824

Description:

If the _isVerifyAmountOutMin variable within the _prevalidateAndCheckSwap() function is set to false, the transaction will revert if the input path requires a swap. Therefore, if a user calls the addTrailingStop() function, which in turn calls the _prevalidateAndCheckSwap() function with a path that requires a swap and the _isVerifyAmountOutMin variable set to false, the transaction will consistently revert.

When calling addTrailingStop() function with a path that requires a swap, the _prevalidateAndCheckSwap always revert because _isVerifyAmountOutMin is false.

```
function addTrailingStop(
    bool _isLong,
    uint256 _posId,
    uint256[] memory _params,
    address[] memory _path
) external payable override nonReentrant {
    //...
    _prevalidateAndCheckSwap(_path, 0, false);
    //...
}

function _prevalidateAndCheckSwap(
    address[] memory _path,
    uint256 _amountOutMin,
    bool _isVerifyAmountOutMin
) internal view returns (bool) {
    //...
    bool shouldSwap = settingsManager.validateCollateralPathAndCheckSwap(_path);

    if (shouldSwap && (_path.length > 2)) {
        //Invalid amountOutMin/swapRouter
        require(_isVerifyAmountOutMin && _amountOutMin > 0 && address(swapRouter) !=
address(0), "IVLAOM/SR");
    }

    return shouldSwap;
}
```

RECOMMENDATION

Should place the `_isVerifyAmountOutMin` variable within the if-condition. The code can be fixed as below:

```
function _prevalidateAndCheckSwap(
    address[] memory _path,
    uint256 _amountOutMin,
    bool _isVerifyAmountOutMin
) internal view returns (bool) {
    //...
    bool shouldSwap = settingsManager.validateCollateralPathAndCheckSwap(_path);

    if (shouldSwap && (_path.length > 2) && _isVerifyAmountOutMin) {
        require(_amountOutMin > 0 && address(swapRouter) != address(0), "IVLAOM/SR");
    }

    return shouldSwap;
}
```

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.2. Abuse transfer LP token to bypass cooldown duration check CRITICAL

Positions:

- `core/Vault.sol`

Description:

In the vault contract, when a user stake the LP tokens will be minted. If the user want to unstake, the user must wait for a cooldown duration passed. Unfortunately, the LP token is transferable so users can transfer token to another address and unstake within passed duration check.

```
function stake(address _account, address _token, uint256 _amount) external nonReentrant {
    //...
    IMintable(ROLP).mint(_account, mintAmount); // Mint LP Token that is transferable
    //...
    emit Stake(_account, _token, _amount, mintAmount);
}

function unstake(address _tokenOut, uint256 _rolpAmount, address _receiver) external
nonReentrant {
    // ...
    require(
        lastStakedAt[msg.sender] + settingsManager.cooldownDuration() <= block.timestamp,
// Bypass check with transfer token
            "Cooldown duration not yet passed"
    );
    //...
    emit Unstake(msg.sender, _tokenOut, _rolpAmount, amountOutInToken);
}
```

RECOMMENDATION

Disable transfer feature or transfer with the stake check.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.3. Decrease pool amount must be in USD unit CRITICAL

Positions:

- `core/PositionHandler.sol`#411

Description:

When increase or decrease pool amount in Vault contract, the token must be calculated in USD unit. In the below code, the contract use amount token instead of `amountInUSD` that calculated before.

```
function _addOrRemoveCollateral(
    bytes32 _key,
    uint256 _txType,
    uint256 _amountIn,
    address[] memory _path,
    uint256[] memory _prices,
    Position memory _position
) internal {
    uint256 prevCollateral = _position.collateral;
    uint256 amountInUSD;
    (amountInUSD, _position) = vaultUtils.validateAddOrRemoveCollateral(
        _amountIn,
        _txType == ADD_COLLATERAL ? true : false,
        _getLastPath(_path), //collateralToken
        _getFirstParams(_prices), //indexPrice
        _getLastParams(_prices), //collateralPrice
        _position
    );

    positionKeeper.unpackAndStorage(_key, abi.encode(_position), DataType.POSITION);

    if (_txType == ADD_COLLATERAL) {
        //...
    } else {
        vault.takeAssetOut(
            _key,
            _position.owner,
            0, //Zero fee for removeCollateral
            _amountIn,
            _getLastPath(_path),
            _getLastParams(_prices)
        );

        vault.decreasePoolAmount(_getLastPath(_path), _amountIn); //use amountInUSD here
        vault.increaseGuaranteedAmount(_getLastPath(_path), prevCollateral -
_position.collateral);
    }

    //...
}
```

RECOMMENDATION

Use `amountInUSD` instead.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.4. Users will lose their assets if the swap fails in opening a new position CRITICAL

Positions:

- `PositionRouter.sol/openNewPosition`

Description:

Whenever a user opens a new position, their assets will be transferred to the Vault through the `_transferAssetToVault()` function. The `_processSwap()` function will return false if the swap fails, but only transactions other than opening a new position with specific parameters for that purpose will be logged as reverted transactions. Therefore, if a user opens a new position and the swap fails (e.g: `amountOut` < `amountOutMin`), the user will lose money, and the transaction will only halt at transferring the user's assets into the Vault.

```
function _modifyPosition(
//  ...
) internal {
//  ...
    //Transfer collateral to vault if required
    if (_isTakeAssetRequired) {
        require(amountIn > 0 && _path.length > 1, "IVLCA/P"); //Invalid collateral
amount/path
        _transferAssetToVault(
            position.owner,
            _path[1],
            amountIn,
            _key,
            _txType
        );
    }

    if (_shouldSwap && _isFastExecute) {
        bool isSwapSuccess;
        address collateralToken;

        {
            (isSwapSuccess, collateralToken, amountIn) = _processSwap( //swap fail here
                _key,
                position.owner,
                _txType,
                amountIn,
                _path,
                _prices,
                params.length > 0 ? _getLastParams(params) : 0
            );

            if (!isSwapSuccess) {
                if (!(_isOpenPosition(_txType) && _isOpenPositionData(_data))) { // Not in
case open position
                    _revertExecute(
                        _key,
                        _txType,
                        params,
                        _prices,
                        _path,
                        "SWF" //Swap failed
                    );
                }

                return;
            }
        }
    }
    ...
}
function _processSwap(
    bytes32 _key,
    address _account,
    uint256 _txType,
    uint256 _pendingCollateral,
    address[] memory _path,
    uint256[] memory _prices,
    uint256 _amountOutMin
```

```
) internal returns (bool, address, uint256) {
    uint256 tokenOutPrice = _prices.length == 0 ? 0 : _getLastParams(_prices);
    require(tokenOutPrice > 0, "IVLTP"); //Invalid token price

    try swapRouter.swapFromInternal(
        _account,
        _key,
        _txType,
        _pendingCollateral,
        _amountOutMin,
        _path
    ) returns (address tokenOut, uint256 swapAmountOut) {
        uint256 swapAmountOutInUSD = priceManager.fromTokenToUSD(tokenOut, swapAmountOut,
tokenOutPrice);
        require(swapAmountOutInUSD > 0, "IVLSAP");
        return (true, tokenOut, swapAmountOutInUSD);
    } catch {
        return (false, _path[1], _pendingCollateral); //revert catch
    }
}
```

RECOMMENDATION

Return the money to the user or revert the transaction.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.5. Arbitrary swap for any account without permission CRITICAL

Positions:

- `swap/SwapRouter.sol`#L171

Description:

In the `swap()` function, users pass `_account` param without approval check. This vulnerability allows for the potential execution of arbitrary swaps on any account. Exploiting this vulnerability, an attacker could engage in a price manipulation attack.

```
function swap(
    address _account,
    address _receiver,
    uint256 _amountIn,
    uint256 _amountOutMin,
    address[] memory _path
) external override nonReentrant returns (bytes memory) {
    // There is no any restricted or approval check for _account and msg.sender
}
```

RECOMMENDATION

High recommendation is remove `_account` in param list and `_account` is declare in function with `msg.sender` value.

If the contract would like delegating swap, it must have delegate check mechanism.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.6. Must update token balance after transfer in Vault CRITICAL

Positions:

- `swap/SwapRouter.sol`#L294

Description:

When the contract directly transfer asset into the `vault`, the `tokenBalances` state is not updated. This missing lead to wrong price calculate in the future.

```
function _vaultSwap(
    address _tokenIn,
    uint256 _amountIn,
    address _tokenOut,
    address _sender,
    address _receiver,
    bool _hasPaid
) internal returns (uint256, bool) {
    if (!_hasPaid) {
        _transferFrom(_tokenIn, _sender, _amountIn);
        _transferTo(_tokenIn, _amountIn, address(vault));
       // Miss update balance
    }
}
```

RECOMMENDATION

The code can be fixed as below:

```
function _vaultSwap(
    address _tokenIn,
    uint256 _amountIn,
    address _tokenOut,
    address _sender,
    address _receiver,
    bool _hasPaid
) internal returns (uint256, bool) {
    if (!_hasPaid) {
        _transferFrom(_tokenIn, _sender, _amountIn);
        _transferTo(_tokenIn, _amountIn, address(vault));
        vault.updateBalance(_tokeIn);
    }
}
```

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.7. The vault swap logic cannot execute because of permission restricted CRITICAL

Positions:

- `swap/SwapRouter.sol`#L408

Description:

In the swap router, vault swap take asset out in the end of function. Specifically, the contract call `takeAssetOut()` function in the vault. However, `takeAssetOut()` restrict only called by `PositionHandler`. This wrong cause the break the logic business execution of contract.

```
function takeAssetOut(
    bytes32 _key,
    address _account,
    uint256 _fee,
    uint256 _usdOut,
    address _token,
    uint256 _tokenPrice
) external override {
    _isPositionHandler(msg.sender, true); // Restricted -> swap router cannot call
}
function _isPositionHandler(address _caller, bool _raise) internal view returns (bool) {
    bool res = _caller == address(positionHandler);

    if (_raise && !res) {
        revert("Forbidden: Not positionHandler");
    }

    return res;
}
```

RECOMMENDATION

The vault must allow `SwapRouter` to call.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.8. Use `tx.origin` for authentication is vulnerable to phishing HIGH

Positions:

- `access/BaseAccess.sol`#L13

Description:

Smart contracts that use `tx.origin` for authorization are vulnerable to phishing attacks, in which a malicious contract can fool the "Has Access User" into running a function and bypass the `limitAccess` modifier check.

```
modifier limitAccess {
    require(hasAccess[tx.origin] || hasAccess[msg.sender], "A:FBD");
    _;
}
```

RECOMMENDATION

The `msg.sender` should be used instead of `tx.origin` for authentication purposes.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.9. The executor's fund will be drained if there are too many "take profit/stop lost" orders placed HIGH

Positions:

- `TriggerOrderManagerV2.sol`#L232

Description:

When calling the `updateTriggerOrders()` function, If user sets to many `_tpPrices` or `_slPrices`, the Executor trigger too many times. As a result of the fee being less than what was charged for gas execution, the executor's money will be drained.

The root cause is the code only gets the fee once, despite the user placing several orders.

```
function updateTriggerOrders(
    address _indexToken,
    bool _isLong,
    uint256 _posId,
    uint256[] memory _tpPrices,
    uint256[] memory _slPrices,
    uint256[] memory _tpAmountPercents,
    uint256[] memory _slAmountPercents,
    uint256[] memory _tpTriggeredAmounts,
    uint256[] memory _slTriggeredAmounts
) external payable nonReentrant {
    ...

    if (triggerOrders[key].tpPrices.length + triggerOrders[key].slPrices.length <
_tpPrices.length + _slPrices.length) {
        require(msg.value == settingsManager.triggerGasFee(), "Invalid triggerGasFee"); //
INCORRECT
    }

    ...
}
```

RECOMMENDATION

The code should calculate triggerGasFee corresponding batch of orders to prevent losing money.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.10. Get first index of path at index 0 instead of index 1 HIGH

Positions:

- `swap/SwapRouter.sol`#L183

Description:

In the `swap()` function, the contract extracts the path to get `tokenIn` and `tokenOut`. However, the contract makes a mistake and gets the first index of the path at index 1. As a result, the function cannot be executed when the user swaps a path token.

```
function swap(
    address _account,
    address _receiver,
    uint256 _amountIn,
    uint256 _amountOutMin,
    address[] memory _path
) external override nonReentrant returns (bytes memory) {
    //...
    {
        tokenIn = _path[1];
        tokenOut = _path[_path.length - 1];
    }
    //...
}
```

RECOMMENDATION

The code can be fixed as above:

```
function swap(
    address _account,
    address _receiver,
    uint256 _amountIn,
    uint256 _amountOutMin,
    address[] memory _path
) external override nonReentrant returns (bytes memory) {
    //...
   require(path.length == 2, "Insufficient length of path");
    {
        tokenIn = _path[1];
        tokenOut = _path[_path.length - 1];
    }
    //...
}
```

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.11. `_isFastExecute` is always true in `triggerPosition()` function MEDIUM

Positions:

- `v2/core/PositionRouterV2.sol`#triggerPosition()

Description:

Only the `triggerOrderManager` may invoke the `triggerPosition()` function. Because of this, the value of the `_isFastExecute` parameter is always `true`.

```
//file: PositionRouter.sol

    function triggerPosition(
    bytes32 _key,
    bool _isFastExecute,
    uint256 _txType,
    address[] memory _path,
    uint256[] memory _prices
) external override {
    require(msg.sender == address(triggerOrderManager), "FBD"); //Forbidden
    require(positionKeeper.getPositionOwner(_key) != address(0), "Position notExist");
    _modifyPosition(
        _key,
        _txType,
        false,
        false,
        msg.sender == address(triggerOrderManager) ? true : _isFastExecute, // ALWAYS TRUE
        _path,
        _prices,
        abi.encode(_getParams(_key, _txType))
    );
}
```

Additionally, the `triggerPosition()` function in the `TriggerOrderManagerV2` calls the above function with a dynamic `_isFastExecute`:

```
//file: TriggerOrderManagerV2.sol

function triggerPosition(
    address _account,
    address _indexToken,
    bool _isLong,
    uint256 _posId
) external nonReentrant {
    ...

    if (position.size == 0) {
        //Trigger for creating position
       ...
        (isFastExecute, prices) = _getPricesAndCheckFastExecute(path);

        if (!isFastExecute) {
            revert("This delay position trigger has already pending");
        }
    }  else {
        //Trigger for addTrailingStop or updateTriggerOrders
        ...
        (isFastExecute, prices) = _getPricesAndCheckFastExecute(path);
        ...

        if (txType == ADD_TRAILING_STOP && !isFastExecute) {
            revert("This trigger has already pending");
        }
    }

    ...
    IPositionRouter(router).triggerPosition(
        key,
        isFastExecute, // DYNAMIC PARAM
        txType,
        path,
        prices
    );
}
```

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.12. No Storage Gap for Upgradeable Contract Might Lead to Storage Slot Collision MEDIUM

Positions:

- `v2/base/*.sol`

Description:

For upgradeable contracts, there must be storage gap to "allow developers to freely add new state variables in the future without compromising the storage compatibility with existing deployments". Otherwise, it may be very difficult to write new implementation code. Without storage gap, the variable in child contract might be overwritten by the upgraded base contract if new variables are added to the base contract. This could have unintended and very serious consequences of the child contracts.

Refer to the bottom part of this article: *https://docs.openzeppelin.com/upgrades-plugins/1.x/writing-upgradeable*

RECOMMENDATION

Recommend adding appropriate storage gap at the end of upgradeable contracts such as the below. Please reference OpenZeppelin upgradeable contract templates.

```
uint256[50] private __gap;
```

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.13. Possible reentrancy in all external function MEDIUM

Positions:

- `staking/StakeRolp.sol`#compound()
- `staking/StakeRolp.sol`#deposit()
- `staking/StakeRolp.sol`#claim()
- `staking/StakeRolp.sol`#withdraw()
- `staking/StakeDualToken.sol`#compound()
- `staking/StakeDualToken.sol`#deposit()
- `staking/StakeDualToken.sol`#claim()
- `staking/StakeDualToken.sol`#withdraw()

Description:

All above functions are external visibility that allow user to call. The critical issues of them are state updating after transferring in claim and withdraw action. This risk is abused execute reentrancy attack by attacker if the token used is ERC1155 (standard include ERC20 but within callback user).

```
function claim(bool[] calldata _isClaim) external {
    UserInfo storage user = userInfo[msg.sender];
    updatePool();
    for(uint i=0; i<rewardInfo.length; i++) {
        PendingReward storage pendingReward =
rewardPending[msg.sender][rewardInfo[i].rwToken];
        uint256 pending =
((user.amountRosx.add(user.amountERosx).add(user.point)).mul(rewardInfo[i].accTokenPerShare
).div(1e18)).sub(pendingReward.rewardDebt);
        if(_isClaim[i]) {
            uint256 claimAmount = pendingReward.rewardPending.add(pending);
            if (claimAmount> 0) {
                if(claimFee > 0) {
                    uint256 fee = (claimFee.mul(claimAmount)).div(100000);
                    claimAmount = claimAmount.sub(fee);
                    rewardInfo[i].rwToken.transfer(feeAddr, fee);
                }
                rewardInfo[i].rwToken.transfer(msg.sender, claimAmount); // Attacker can
control logic at this line before rewardPending is assigned to zero
                pendingReward.rewardPending = 0;
            }
        } else {
            pendingReward.rewardPending = pendingReward.rewardPending.add(pending);
        }
        pendingReward.rewardDebt =
((user.amountRosx.add(user.amountERosx).add(user.point)).mul(rewardInfo[i].accTokenPerShare
)).div(1e18);
    }
}
```

RECOMMENDATION

The contract should use `ReentrancyGuard` from @Openzeppelin library protect all external/public functions.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

2.2.14. Incorrect position data validation MEDIUM

Positions:

- `VaultUtilsV2.sol`#validatePositionData()

Description:

The validation function uses the same block to validate both long and short positions. Additionally, the validation process does not check the prices of limit, stop, or stop-limit orders corresponding to long or short positions.

```
function validatePositionData(
    bool _isLong,
    address _indexToken,
    OrderType _orderType,
    uint256 _indexTokenPrice,
    uint256[] memory _params,
    bool _raise
) external view override returns (bool) {
    if (_raise && _params.length != 8) {
        revert("Invalid params length, must be 8");
    }

    bool orderTypeFlag;
    uint256 marketSlippage;

    if (_params[5] > 0) {
        uint256 indexTokenPrice = _indexTokenPrice == 0 ?
priceManager.getLastPrice(_indexToken) : _indexTokenPrice;

        if (_isLong) {
            if (_orderType == OrderType.LIMIT && _params[2] > 0) {
                orderTypeFlag = true;
            } else if (_orderType == OrderType.STOP && _params[3] > 0) {
                orderTypeFlag = true;
            } else if (_orderType == OrderType.STOP_LIMIT && _params[2] > 0 && _params[3] >
0) {
                orderTypeFlag = true;
            } else if (_orderType == OrderType.MARKET) {
                marketSlippage = _getMarketSlippage(_params[1]);
                checkSlippage(_isLong, _getFirstParams(_params), marketSlippage,
indexTokenPrice);
                orderTypeFlag = true;
            }
        } else {
            if (_orderType == OrderType.LIMIT && _params[2] > 0) {
                orderTypeFlag = true;
            } else if (_orderType == OrderType.STOP && _params[3] > 0) {
                orderTypeFlag = true;
            } else if (_orderType == OrderType.STOP_LIMIT && _params[2] > 0 && _params[3] >
0) {
                orderTypeFlag = true;
            } else if (_orderType == OrderType.MARKET) {
                marketSlippage = _getMarketSlippage(_params[1]);
                checkSlippage(_isLong, _getFirstParams(_params), marketSlippage,
indexTokenPrice);
                orderTypeFlag = true;
            }
        }
    } else {
        orderTypeFlag = true;
    }

    if (_raise) {
        require(orderTypeFlag, "Invalid positionData");
    }

    return (orderTypeFlag);
}
```

RECOMMENDATION

Should validate position data with more detail.

UPDATES

- *Oct 11, 2023*: This issue has been acknowledged and fixed by the Roseon team.

3. VERSION HISTORY

Version	Date	Status/Change	Created by
1.0	*Sep 05, 2023*	Private Report	Verichains Lab
1.1	*Oct 11, 2023*	Public Report	Verichains Lab

Table 3. Report versions history