CD SECURITY

AUDIT REPORT

Volt Staking
January 2025

Prepared by
GT_GSEC
Hals
yotov721

Introduction

A time-boxed security review of the **Volt Staking** protocol was done by **CD Security**, with a focus on the security aspects of the application's implementation.

Disclaimer

A smart contract security review can never verify the complete absence of vulnerabilities. This is a time, resource, and expertise-bound effort where we try to find as many vulnerabilities as possible. We can not guarantee 100% security after the review or even if the review will find any problems with your smart contracts. Subsequent security reviews, bug bounty programs, and on-chain monitoring are strongly recommended.

About **Volt Staking**

VOLT's staking allows users to stake tokens and earn rewards in VOLT based on the time staked. There are 4 pools with different cycles times each ranging 1 to 4 weeks. If the user stake duration is greater than the pool cycle the user would receive rewards from that pool for that cycle.
VOLT offers a time bound staking range and bonus rewards based on the time staked. The rewards are collected from Uni V3 positions providing liquidity of VOLT pairs. When pool rewards are claimed they are all converted to VOLT and distributed to staking pools.

Severity classification

Severity	**Impact: High**	**Impact: Medium**	**Impact: Low**
Likelihood: High	Critical	High	Medium
Likelihood: Medium	High	Medium	Low
Likelihood: Low	Medium	Low	Low

Impact - the technical, economic, and reputation damage of a successful attack

Likelihood - the chance that a particular vulnerability gets discovered and exploited

Severity - the overall criticality of the risk

Security Assessment Summary

***review commit hash* - 7a059273795c80449d620c38f173b34815655266**

Scope

The following smart contracts were in scope of the audit:

- src/*

The following number of issues were found, categorized by their severity:

- High: 0
- Medium: 5
- Low: 2
- Info: 7

Findings Summary

ID	Title	Severity	Resolution
[M-01]	No slippage application in the `_swapToVolt` function	Medium	Fixed
[M-02]	No slippage application in the `_addLiquidityUniV3` function	Medium	Fixed
[M-03]	Missing authorization in the `collectFees` can lead to sandwich	Medium	Fixed
[M-04]	Undistributed rewards may be stuck in the contract if no stakers exist in a cycle	Medium	Fixed
[M-05]	Swap deadline set to `block.timestamp` provides no protection	Medium	Acknowledged
[L-01]	The treasury address is defined as immutable	Low	Acknowledged
[L-02]	Lack of withdrawal mechanism for deposited NFT liquidity positions	Low	Acknowledged
[I-01]	Protocol does not support non-standard ERC-20 tokens	Info	Fixed
[I-02]	Redundant assertions within the `_poolCalculateCurrentCycleId` function	Info	Acknowledged
[I-03]	Liquidity over the full price range can lead to lower fees earning	Info	Acknowledged
[I-04]	Max allowance applied within the `_setAllowanceMaxIfNeeded` function	Info	Acknowledged
[I-05]	Check effect interaction pattern violation	Info	Acknowledged
[I-06]	Possible gas over consumption within the `_isFullRangeNftUniV3` function	Info	Acknowledged
[I-07]	Remove unused and debug imports	Info	Fixed

Detailed Findings

[M-01] No slippage application in the `_swapToVolt` function

Severity

Impact: Medium

Likelihood: Medium

Description

The `_swapToVolt` function has no slippage applied (`uint256 amountOutMin = 0;`). This function is used within the `collectFees` function and the `swapToVoltOnly` which is desired for the privileged account. The lack of slippage control can lead directly to the sandwich attacks.

```
    /// @notice Swap `amountIn` amount of `tokenIn` ERC-20 tokens to
`$VOLT` token (represented by `voltToken`)
    /// @dev The swap happens according to `VoltSwapRouterConfig` set for
`tokenId`
    function _swapToVolt(address tokenIn, uint256 amountIn) internal {
        if (amountIn == 0) return;
        address tokenOut = address(voltToken);

        if (tokenIn == address(0)) revert
PositionManagerInvalidErc20Token();
        if (tokenIn == tokenOut) revert
PositionManagerInvalidErc20Token();

        VoltSwapRouterConfig memory config =
_voltSwapRouterConfigs[tokenIn];

        // There is no slippage control
        uint256 amountOutMin = 0;
        address router = config.routerAddress;

        if (config.routerType == VoltSwapRouterType.UniV2) {
            _setAllowanceMaxIfNeeded(IERC20(tokenIn), amountIn,
address(router));
            _swapUniV2(IRouterV2(router), tokenIn, tokenOut, amountIn,
amountOutMin);
        } else if (config.routerType == VoltSwapRouterType.UniV3) {
            uint24 feeTier = config.feeTier;
            if (feeTier == 0) revert PositionManagerInvariant("fee tier");
            _setAllowanceMaxIfNeeded(IERC20(tokenIn), amountIn,
address(router));
            _swapUniV3(IRouterV3(router), tokenIn, tokenOut, amountIn,
amountOutMin, feeTier);
        } else {
            revert PositionManagerInvariant("router type");
            }
        }
```

Recommendations

Consider enhancing the `_swapToVolt` function by introducing slippage, that provided by the user as input is preventing large deviation within the pair's swap.

[M-02] No slippage application in the `_addLiquidityUniV3` function

Severity

Impact: Medium

Likelihood: Medium

Description

The `_addLiquidityUniV3` function has no slippage applied (`amount0Min: 0, amount1Min: 0,`). This function is used only within the `addLiquidity` which is desired for the privileged account. The adding liquidity process within the Uniswapv3 is prone to the sandwich attacks, as it is stated in the documentation:

> In production, amount0Min and amount1Min should be adjusted to create slippage protections.

```
    function _addLiquidityUniV3(uint256 tokenId, uint256 amountAdd0,
uint256 amountAdd1)
        internal
        returns (uint128 liquidity, uint256 amount0, uint256 amount1)
    {
        UniV3Nft memory nft = _getNftUniV3(tokenId);
        _setAllowanceMaxIfNeeded(IERC20(nft.token0), amountAdd0,
address(nftManager));
        _setAllowanceMaxIfNeeded(IERC20(nft.token1), amountAdd1,
address(nftManager));

        INonfungiblePositionManager.IncreaseLiquidityParams memory params
= INonfungiblePositionManager
            .IncreaseLiquidityParams({
            tokenId: tokenId,
            amount0Desired: amountAdd0,
            amount1Desired: amountAdd1,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        });
        (liquidity, amount0, amount1) =
nftManager.increaseLiquidity(params); //@audit allowance not reduced
    }
```

Recommendations

Consider enhancing the `_addLiquidityUniV3` function by introducing slippage, that provided by the user as input is preventing large deviation within the pair's swap.

[M-03] Missing authorization in the `collectFees` can lead to sandwich

Severity

Impact: Medium

Likelihood: Medium

Description

The `collectFees` function performs following operations for every input token Id:

- It collects fees via `_collectFeesUniV3`.
- It swaps tokens to Volt tokens via `_swapToVolt`.
- Lastly, it transfer Volt tokens into staking contract via `_transferVoltToStakingVault`.

Each of this operation can be done individually respectively via `collectFeesOnly`, `swapToVoltOnly`, `transferToVaultOnly` functions. All of these three functions are protected by the `onlyRole(DEFAULT_ADMIN_ROLE)` modifier, whereas the `collectFees` has no authorisation applied at all.

The `_swapToVolt` function has no slippage applied (`uint256 amountOutMin = 0;`). Thus, the swap will be done for any possible ratio between pair's tokens.

A malicious user can leverage this fact and trigger `collectFees` to sandwich it and manipulate the price of pair before fees collection and after to collect potential profits.

However, the success of attack depends on several factors, including: whether Uniswapv2 or Uniswapv3 is targeted, what is the liquidity provided to the Uniswapv3's pool, what is the amount of fee collected by the protocol in prior of the swap.

```
    /// @notice Collect fees, swap to `$VOLT` tokens, and transfer to
`IStakingVault` instance
    function collectFees(uint256[] calldata tokenIds) external {
        if (tokenIds.length > MAX_COLLECT_FEES_BATCH_SIZE) revert
PositionManagerArrayTooLong();

        for (uint256 i = 0; i < tokenIds.length; i++) {
            uint256 tokenId = tokenIds[i];
            if (!_uniV3NftIdIsManaged[tokenId]) revert
PositionManagerUniswapV3PositionNotManaged(tokenId);

            (uint256 amount0, uint256 amount1) =
_collectFeesUniV3(tokenId);
            emit PositionFeesCollected(tokenId);

            UniV3Nft memory nft = _getNftUniV3(tokenId);

            if (nft.token0 != address(voltToken) && amount0 != 0) {
                _swapToVolt(nft.token0, amount0);
                emit TokenToVoltSwapped(nft.token0);
            }

            if (nft.token1 != address(voltToken) && amount1 != 0) {
                _swapToVolt(nft.token1, amount1);
                emit TokenToVoltSwapped(nft.token1);
            }
        }

        _transferVoltToStakingVault();
    }
```

Recommendations

Consider enhancing the `collectFees` function either with the authorisation done for privileged account only or by introducing internal slippage preventing large deviation within swaps.

[M-04] Undistributed rewards may be stuck in the contract if no stakers exist in a cycle

Severity

Impact: Medium

Likelihood: Medium

Description

In the `StakingVault` contract, rewards are received from the `PositionManager` via `acceptPositionManagerRewards()`. These rewards are then distributed **equally among pools for the current cycle**.

However, if **no stakers exist in a particular cycle**, the rewards for that cycle **will remain stuck in the contract**, as the undistributed rewards from cycles without stakers are not migrated to the next cycle, unlike cycle shares, which are migrated when a new cycle opens via `endCycleIfNeeded()`.

Recommendation

Implement a `sweep()` function to rescue stuck rewards from cycles with no stakers.

[M-05] Swap deadline set to `block.timestamp` provides no protection

Severity

Impact: Medium

Likelihood: Medium

Description

In the `PositionManager` contract, swaps are executed using either the Uniswap V2 or V3 router, depending on the configuration. When a swap is executed via Uniswap V2 router, the **deadline parameter**, which should ensure that transactions expire after a certain time, is **incorrectly set to `block.timestamp`**:

```
function _swapUniV2(
        IRouterV2 _router,
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 amountOutMinimum
    ) internal {
     //...
        _router.swapExactTokensForTokensSupportingFeeOnTransferTokens(
            amountIn, amountOutMinimum, path, address(this),
block.timestamp
        );
    }
```

Since `block.timestamp` represents the **current block time**, it does not provide **any protection** against delayed or stale transaction execution. This means:

- Transactions stuck in the mempool can be executed at any future time when the price has changed significantly.
- If the transaction is executed later due to network congestion, the received token amount may be much lower than expected, leading to funds loss.

Same issue exists when adding liquidity to an existing deposited UniV3 liquidity position:

```
    function _addLiquidityUniV3(uint256 tokenId, uint256 amountAdd0,
uint256 amountAdd1)
        internal
        returns (uint128 liquidity, uint256 amount0, uint256 amount1)
    {
        //...
        INonfungiblePositionManager.IncreaseLiquidityParams memory params
= INonfungiblePositionManager
            .IncreaseLiquidityParams({
            tokenId: tokenId,
            amount0Desired: amountAdd0,
            amount1Desired: amountAdd1,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        });
        (liquidity, amount0, amount1) =
nftManager.increaseLiquidity(params);
    }
```

Recommendation

Instead of hardcoding `block.timestamp` as the deadline, introduce a **caller-specified deadline parameter** to ensure that the swap is executed within an acceptable timeframe.

[L-01] The treasury address is defined as immutable

Severity

Impact: Medium

Likelihood: Medium

Description

The `treasury` address is defined as immutable within the protocol. This address is used to collect fees within the `acceptPositionManagerRewards` function. Additionally, the protocol makes use of contracts that are not upgradable. This approach can be risky, as whenever the `treasury` address is compromised, there is no possibility to update the treasury address. Moreover, new deployment of the solution might be cumbersome, especially that the `PositionManager` stores the Uniswap's positions.

```
...
    address public immutable treasury;
...
```

Recommendations

Consider making the 'treasury' address non-immutable and implement functionality that allows the protocol's owner to update the address whenever it is compromised.

[L-02] Lack of withdrawal mechanism for deposited NFT liquidity positions

Severity

Impact: Medium **Likelihood:** Low

Description

In the `PositionManager` contract, Uniswap V3 NFT liquidity positions are deposited by the admin, but the contract does not provide a function to withdraw or reclaim these positions. Additionally, it lacks functionality to decrease the liquidity of a deposited position.

Without a withdrawal mechanism, once an NFT liquidity position is deposited, it becomes permanently locked within the contract, restricting liquidity management and preventing repositioning.

Recommendations

Introduce a withdrawal function that allows the admin to retrieve deposited liquidity positions. Additionally, consider adding functionality to partially decrease liquidity from existing positions.

[I-01] Protocol does not support non-standard ERC-20 tokens

Description

The `StakingVault` operates on ERC20 tokens performing specifically `transfer` and `transferFrom` operations. The protocol owner confirmed that currently supported tokens are: $SURGE, $SHIB and $VOLT, however, any other token can be used as staking token. However, some of the ERC20 tokens does not follow EIP-20 standard. E.g. the USDT token does not return boolean value upon successful call to the `transfer` and `transferFrom` functions, instead it returns void. Thus it is not compatible with the `IERC20` interface. Any attempt to call for such non-standard ERC20 token will result in transaction revert.

```
    /// @notice Address of the ERC20 token used for staking
    IERC20 public immutable stakingToken;
    /// @notice Address of the ERC20 token used for rewards
    IERC20 public immutable rewardsToken;
```

Recommendations

Consider enhancing the implementation by means of the OpenZeppelin's `SafeERC20` library that provides functionality to handle both standard and non-standard ERC20 tokens, so any token type could be used as staking or reward token.

[I-02] Redundant assertions within the `_poolCalculateCurrentCycleId` function

Description

The `_poolCalculateCurrentCycleId` function has two assertions implemented and both appears to be redundant. First assertion is duplicated within the `endCycleIfNeeded` which calls the `_poolCalculateCurrentCycleId` internally. Second assertion cannot achieve negative result as `cycleDurationDays` for every pool info is defined within the constructor with one of the values: 7, 14, 21, 28.

```
    /// @notice Update the `currentCycleId` in all reward pools, if the
previous cycles have ended
    function endCycleIfNeeded() public {
        uint256 currentTime = block.timestamp;
        if (currentTime < rewardPoolsStartAt) revert
StakingVaultRewardPoolsNotStarted();

        for (uint256 i; i < REWARD_POOL_COUNT; i++) {
            uint256 oldCurrentCycleId = _rewardPools[i].currentCycleId;
            uint256 newCurrentCycleId =
_poolCalculateCurrentCycleId(_rewardPools[i], currentTime);
            if (newCurrentCycleId > oldCurrentCycleId) {
                _rewardPools[i].currentCycleId = newCurrentCycleId;

                uint256 poolId = _rewardPools[i].id;
                _rewardPoolShares[poolId][newCurrentCycleId] =
_rewardPoolShares[poolId][oldCurrentCycleId];
                emit PoolCycleChanged(poolId, oldCurrentCycleId,
newCurrentCycleId);
            }
        }
    }

    /// @notice Calculate the `cycleId` of a pool, based on `currentTime`
    function _poolCalculateCurrentCycleId(RewardPoolInfo memory poolInfo,
uint256 currentTime)
        private
        view
        returns (uint256)
    {
        if (currentTime < rewardPoolsStartAt) revert
StakingVaultRewardPoolsNotStarted();
        if (poolInfo.cycleDurationDays == 0) revert
StakingVaultInvalidCycleDuration();

        uint256 firstCycleId = 1;

        uint256 cycleDuration = poolInfo.cycleDurationDays * 1 days;
        uint256 diffDuration = currentTime - rewardPoolsStartAt;
        uint256 diffCycleId = diffDuration / cycleDuration;
        uint256 cycleId = diffCycleId + firstCycleId;

        return cycleId;
    }
```

Recommendations

Consider removing redundant assertions to save some Gas during transaction execution.

[I-03] Liquidity over the full price range can lead to lower fees earning

Description

The `depositPosition` function enforces that position provided must be in full price range (via the `_isFullRangeNftUniV3` function). However, as it is stated within the Uniswap's documentation such approach can lead to lowered profits, comparing to the concentrated positions.

> It is important to understand that token prices may not reach all these price levels. Wide ranges can earn lower fees compared to concentrated positions in narrower price ranges where trading is more likely to occur.

```
    function depositPosition(uint256 tokenId) external nonReentrant
onlyRole(DEFAULT_ADMIN_ROLE) {
        UniV3Nft memory nft = _getNftUniV3(tokenId);
        _validateNftUniV3(nft);

        // Check position NFT is full range
        if (!_isFullRangeNftUniV3(nft)) revert
PositionManagerInvalidUniswapV3Position(tokenId);
...
```

```
    function _isFullRangeNftUniV3(UniV3Nft memory nft) internal view
returns (bool) {
        int24 tickSpacing = _feeAmountTickSpacing[nft.fee];
        if (tickSpacing == 0) revert UniswapHelperZeroTickSpacing();

        int24 tickMax = (TickMath.MAX_TICK / tickSpacing) * tickSpacing;
        int24 tickMin = (TickMath.MIN_TICK / tickSpacing) * tickSpacing;

        return nft.tickLower <= tickMin && nft.tickUpper >= tickMax;
    }
...
```

Recommendations

Review the protocol defined business rules, design and implementation and consider whether concentrated positions could be more profitable and in favour of the protocol purpose, comparing to the full price range positions.

[I-04] Max allowance applied within the `_setAllowanceMaxIfNeeded` function

Description

The _setAllowanceMaxIfNeeded function sets allowance to the maximum value allowing infinite allowance once for every liquidity increase, swap or transfer to staking contract operation. This setting is then not reset anywhere later. While such approach might be convenient it shifts the risk to the allowance consuming contracts and their safety. Setting max allowance is considered as a deviation from leading security practices.

```
    function _setAllowanceMaxIfNeeded(IERC20 token, uint256 amount,
address spender) internal {
        uint256 allowance = token.allowance(address(this), spender);
        if (allowance >= amount) return;
        token.approve(spender, type(uint256).max);
    }
...
```

Recommendations

Review the protocol defined business rules, design and implementation and consider implementing singular allowance increase before performing transfer from operation and zeroing the allowance afterwards.

[I-05] Check effect interaction pattern violation

Description

It was identified that the `_stake` and `_unstake` functions do not follow the Check-Effect-Interaction (CEI) pattern. Specifically, within the `_unstake` the `stakingToken` transfer is being done before the `_userStakeInfo` record is deleted.

The CEI pattern is commonly used in Solidity to prevent any kind of reentrancy attacks. While reentrancy is mainly possible for native asset tokens transfer, if only the ERC20 token implements hooks, it might be vulnerable to reentrancy as well. Such known example are ERC777 tokens. However, no such tokens usage were identified within the assessment. The finding is reported as a deviation from leading security practices.

```
    function _unstake(address user) internal returns (StakeInfo memory) {
        endCycleIfNeeded();

        StakeInfo memory info = _userStakeInfo[user];
        if (info.lockedAt == 0) revert StakingVaultUserNotStaking();

        uint256 unlockTimestamp = info.lockedAt +
(uint256(info.lockedDays) * 1 days);
        if (unlockTimestamp > block.timestamp) revert
StakingVaultUserNotFinishedStaking();

        uint256 amount = info.amount;

        // Transfer staking tokens from the contract to the user
        // And remove the old stake entry for the user
        bool success = stakingToken.transfer(user, amount);
        if (!success) revert
StakingVaultErc20TransferFailed(address(stakingToken), msg.sender, user,
amount);
        delete _userStakeInfo[user];
...
```

Recommendations

It is recommended to follow the CEI pattern in every case.

[I-06] Possible gas over consumption within the `_isFullRangeNftUniV3` function

Description

The `UniswapHelper` contract defines the `_feeAmountTickSpacing` collection which is by default filled with 3 records within the constructor. Then, this collection is being used in the `_isFullRangeNftUniV3` function to verify the `UniV3Nft` fee. Also, two arithmetic operations, which results are constant, are being executed. This approach actually consumes more Gas, than calculating the `tickMin` and `tickMax` off-chain and storing them instead of the `_feeAmountTickSpacing`.

```
...
    mapping(uint24 => int24) internal _feeAmountTickSpacing;
...
    constructor(INonfungiblePositionManager _nftManager) {
        nftManager = _nftManager;

        // See: https://github.com/Uniswap/v3-
core/blob/main/contracts/UniswapV3Factory.sol
        _feeAmountTickSpacing[500] = 10;
        _feeAmountTickSpacing[3000] = 60;
        _feeAmountTickSpacing[10000] = 200;
    }
...
    function _isFullRangeNftUniV3(UniV3Nft memory nft) internal view
returns (bool) {
        int24 tickSpacing = _feeAmountTickSpacing[nft.fee];
        if (tickSpacing == 0) revert UniswapHelperZeroTickSpacing();
        int24 tickMax = (TickMath.MAX_TICK / tickSpacing) * tickSpacing;
        int24 tickMin = (TickMath.MIN_TICK / tickSpacing) * tickSpacing;

        return nft.tickLower <= tickMin && nft.tickUpper >= tickMax;
    }
...
```

Recommendations

Consider calculating the `tickMin` and `tickMax` off-chain and use them instead of `_feeAmountTickSpacing`.

[I-07] Remove unused and debug imports

Description

On several places in the code there are unused or debug imports

In `RewardMath`:

```
import {SafeCast} from "@openzeppelin-contracts-
5.1.0/utils/math/SafeCast.sol";

import {MIN_BONUS_PERCENT, MAX_BONUS_PERCENT} from "./constants.sol";
import {MIN_LOCKING_PERIOD_DAYS, MAX_LOCKING_PERIOD_DAYS} from
"./constants.sol";
```

In `StakingVault`:

```
import {console} from "forge-std-1.9.4/src/console.sol";
```

In `UniswapHelper`:

```
import {TransferHelper} from "./third-party/uniswap-v3-periphery-
1.3.0/libraries/TransferHelper.sol";
```

In `PositionManager`:

```
import {IERC721} from "@openzeppelin-contracts-
5.1.0/token/ERC721/IERC721.sol";
```

In `BonusMath`:

```
import {SafeCast} from "@openzeppelin-contracts-
5.1.0/utils/math/SafeCast.sol";
```

Recommendations

Remove unused and debug imports