our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Surge

SECURITY REVIEW

Date: 13 January 2025

CONTENTS

1. About Shieldify

Positioned as the first hybrid Web3 Security company, Shieldify shakes things up with a unique subscription-based auditing model that entitles the customer to unlimited audits within its duration, as well as top-notch service quality thanks to a disruptive 6-layered security approach. The company works with very well-established researchers in the space and has secured multiple millions in TVL across protocols, also can audit codebases written in Solidity, Rust, Go, Vyper, Move and Cairo.

Learn more about us at shieldify.org.

2. Disclaimer

This security review does not guarantee bulletproof protection against a hack or exploit. Smart contracts are a novel technological feat with many known and unknown risks. The protocol, which this report is intended for, indemnifies Shieldify Security against any responsibility for any misbehavior, bugs, or exploits affecting the audited code during any part of the project's life cycle. It is also pivotal to acknowledge that modifications made to the audited code, including fixes for the issues described in this report, may introduce new problems and necessitate additional auditing.

3. About Surge

Powering a Lightning-Fast deflation with Electrifying buy and burn.

Volt, built on TitanX, is a hyper-deflationary token with a unique auction system. It features a capped supply with all tokens distributed in the first 10 days, triggering full deflation afterwards. Volt utilizes 80% of system value to buy tokens + 8% of value for growing bonded liquidity growth. Volt enters deflation quickly with a massive buy and burn.

Learn more about Volt's concept and the technicalities behind it here.

4. Risk Classification

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

4.1 Impact

- **High** - results in a significant risk for the protocol's overall well-being. Affects all or most users
- **Medium** - results in a non-critical risk for the protocol affects all or only a subset of users, but is still unacceptable
- **Low** - losses will be limited but bearable - and covers vectors similar to griefing attacks that can be easily repaired

4.2 Likelihood

- **High** - almost certain to happen and highly lucrative for execution by malicious actors

- **Medium** - still relatively likely, although only conditionally possible
- **Low** - requires a unique set of circumstances and poses non-lucrative cost-of-execution to rewards ratio for the actor

5. Security Review Summary

The security review lasted 5 days with a total of 160 hours dedicated by 4 researchers from the Shieldify team.

Overall, the code is well-written. The audit report highlighted four High-severity issues, three Medium-severity issues, and several Low-severity vulnerabilities, along with informational recommendations. The vulnerabilities mainly stem from the lack of slippage controls, missing key functionalities, and potential malicious scenarios, such as staking for very short periods.

The Surge team has been very responsive to the Shieldify research team's inquiries. Their proactive security approach is evident, as they are conducting multiple security reviews with various companies, showcasing their thoroughness and dedication to securing their platform.

5.1 Protocol Summary

Project Name	**Surge**
Repository	surge-contracts
Type of Project	DeFi, Buy and Burn, Staking
Audit Timeline	5 days

5.2 Scope

The following smart contracts were in the scope of the security review:

File	nSLOC
src/PositionManager.sol	143
src/UniswapHelper.sol	125
src/StakingVault.sol	257
src/BonusMath.sol	33
src/constants.sol	6
src/IStakingVault.sol	3
Total	**567**

6. Findings Summary

The following number of issues have been identified, sorted by their severity:

- **Critical & High** issues: **4**
- **Medium** issues: **2**

- **Low** issues: **2**
- **Informational** issues: **5**

ID	Title	Severity	Status
[H-01]	Lack of Slippage and Access Control in `PositionManager.sol` Can Result in Stolen Funds	High	Fixed
[H-02]	Missing Implementation in `PositionManager.sol` Causes the Liquidity in Positions to Not Be Retrievable	High	Acknowledged
[H-03]	Unstake Causes All Users to Lose Their Rewards	High	Fixed
[H-04]	Users Can Take Their Share of Rewards in Cycles They Stayed for Just a Couple of Seconds	High	Acknowledged
[M-01]	The `onERC721Received()` Is Not Implemented Correctly	Medium	Fixed
[M-02]	Staking and Reward Token Overlap Can Risk User Stakes	Medium	Fixed
[L-01]	The `increaseLiquidity()` in `PositionManager.sol` is Missing Slippage Protection	Low	Acknowledged
[L-02]	The `StakingVault` Does Not Support Fee-On-Transfer Tokens (or Weird Tokens)	Low	Acknowledged
[I-01]	Outstanding Allowances and Max Approvals Could Cause Loss of Funds in Multiple Scenarios	Informational	Fixed
[I-02]	The `userShares()` Does Not Show the Shares of the User Until the User Unstakes	Informational	Fixed
[I-03]	Use `safeERC20` in Order to Support Tokens Such as USDT	Informational	Acknowledged
[I-04]	Typography	Informational	Fixed
[I-05]	The `collectFees()` in `PositionManager.sol` Might Revert	Informational	Fixed

7. Findings

[H-01] Lack of Slippage and Access Control in `PositionManager.sol` Can Result in Stolen Funds

Severity

High Risk

Description

The `PositionManager._swapToVolt()` does not have slippage control and according to code comments, it looks like this is intentional.

Location of Affected Code

File: PositionManager.sol

```
function _swapToVolt(address tokenIn, uint256 amountIn) internal {
    if (amountIn == 0) return;
    address tokenOut = address(voltToken);
    require(tokenIn != address(0), "Invalid token");
    require(tokenIn != tokenOut, "Invalid token");
    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));
        return _swapUniV2(IRouterV2(router), tokenIn, tokenOut, amountIn,
            amountOutMin);
    } else if (config.routerType == VoltSwapRouterType.UniV3) {
        uint24 feeTier = config.feeTier;
        require(feeTier != 0, "Invalid fee tier");
        _setAllowanceMaxIfNeeded(IERC20(tokenIn), amountIn, address(
           router));
        return _swapUniV3(IRouterV3(router), tokenIn, tokenOut, amountIn,
            amountOutMin, feeTier);
    } else {
        revert("Invalid router type");
    }
}
```

Impact

Some could argue that the swaps will be made only via private transactions by the admin in order to avoid sandwich attacks. Still not 100% safe but the issue here is 100% exploitable since `_swapToVolt()` is called inside `collectFees()` which is callable by anybody. So an attacker can extract the value of the accrued fees by making a call for the collection of the accrued fees himself and sandwiching the swap since `amountOutMin` is hardcoded to 0.

Recommendation

Consider adding access control to `collectFees()` or removing it and modifying `_swapToVolt()` to enforce caller specified minimal amount out.

Team Response

Fixed.

[H-02] Missing Implementation in `PositionManager.sol` Causes the Liquidity in Positions to Not Be Retrievable

Severity

High Risk

Description

`PositionManager` can receive liquidity position NFTs, collect fees, and add liquidity to them but there is no way for the admin to transfer these positions outside the contract, access the liquidity they are holding, or burn them.

Location of Affected Code

File: PositionManager.sol

Impact

As a result, the underlying liquidity will not be accessible anymore resulting in a loss of funds. Furthermore, these LP positions will be configured for a specific price range. Once the price goes outside of this price range it is normal for liquidity providers to be able to move the liquidity to another price range that is more profitable by decreasing the liquidity from their current LP position and to create a new position with a more relevant price range.

So over some time when the price moves the admin will not be able to collect fees efficiently as well.

Recommendation

Consider implementing logic for either transferring the LP position out of the contract or logic that will be able to access the underlying liquidity of the LP positions that are in control of the `PositionManager`.

Team Response

Acknowledged.

[H-03] Unstake Causes All Users to Lose Their Rewards

Severity

High Risk

Description

The `StakingVault.unstake()` function should reduce the total amount of shares in the current cycle by the shares that get unstaked by the user. Instead, the total amount of shares from all stakes gets deleted.

Location of Affected Code

File: StakingVault.sol

```
function _unstake(address user) internal returns (StakeInfo memory) {
    // code

    // Decrease total shares and close claimable rewards for every pool
    for (uint256 i; i < REWARD_POOL_COUNT; i++) {
        uint256 poolId = _rewardPools[i].id;
        uint256 cycleId = _rewardPools[i].currentCycleId;

        ClaimableReward storage r = _userRewardPoolClaimableReward[user][
           poolId];
        require(r.cycleStart != 0, "No claimable reward"); // This should
            never happen
        require(r.cycleEnd == 0, "Claimable reward is not active"); //
           This should never happen
        require(r.cycleStart <= cycleId, "Claimable reward is start cycle
            invalid"); // This should never happen

        // Close claimable reward entry - this results in `r.active() ==
           false`
        r.cycleEnd = cycleId;

@>      uint256 shares = _rewardPoolShares[poolId][cycleId];

        // Decrease total shares for current pool cycle
@>      _rewardPoolShares[poolId][cycleId] -= shares;

        // If there are no claims possible for this user in this pool -
           remove the claimable reward entry
        if (r.cycleStart == r.cycleEnd) {
            delete _userRewardPoolClaimableReward[user][poolId];
        }
    }

    return info;
}
```

Impact

Since the `claimRewardsToOwed()` calculates the reward of each user by using `_rewardPoolShares()` this results in the loss of rewards for all users for all future cycles.

Proof of Concept

Place this snipped inside the `StakingVault.unstake.t.sol`:

```
function test_unstake_lose_rewards_poc() public {
    address user = makeAddr("user");
    address attacker = makeAddr("attacker");

    uint256 amount = 1000;
    uint16 period = 315; // days

    vm.startPrank(user);
    deal(address(stakingToken), user, amount);
    stakingToken.approve(address(uut), amount);
    uut.stake(amount, period);
    vm.stopPrank();

    vm.startPrank(attacker);
    deal(address(stakingToken), attacker, amount);
    stakingToken.approve(address(uut), amount);
    uut.stake(amount, 28);

    console.log("total shares  cycle 1: ", uut.totalShares(1, 1));

    skip(28 days + 10);
    console.log("skip 28 days");
    uut.endCycleIfNeeded();

    console.log("total shares  cycle 5: ", uut.totalShares(1, 5));

    uut.unstake();
    console.log("attacker unstakes");

    console.log("total shares  cycle 1: ", uut.totalShares(1, 1));
    console.log("total shares  cycle 5: ", uut.totalShares(1, 5));
    vm.stopPrank();

    uint256 daysUntilUserUnlock = period - 28;
    skip(daysUntilUserUnlock * 1 days);
    console.log("skip days: ", daysUntilUserUnlock);
    uut.endCycleIfNeeded();

    console.log("total shares  c5: ", uut.totalShares(1, 5));
    console.log("total shares  c12: ", uut.totalShares(1, 12));
}
```

Recommendation

In the `unstake()` reduce the amount of the total shares for the current cycle only with the number of shares that are being unstaked.

Team Response

Fixed.

[H-04] Users Can Take Their Share of Rewards in Cycles They Stayed for Just a Couple of Seconds

Severity

High Risk

Description

Let's illustrate how cycles and reward retention function. Below we will see the distribution of rewards in cycles if we call `acceptPositionManagerRewards()` every 7 days with 1000 tokens.

```
// # Rewards in cycles

//              pool 1      pool 2      pool 3      pool 4
// cycle 1      250         500         750         1000
// cycle 2      250         500         750         1000
// cycle 3      250         500         500
// cycle 4      250         500
// cycle 5      250         250
// cycle 6      250
// cycle 7      250
// cycle 8      250
// ...

// # Cycle durations

//          cycles
// Pool 1   [  1 ][  2 ][  3 ][  4 ][  5 ][  6 ][  7 ][  8 ][  9 ][ 10 ][
    11 ][ 12 ]
// Pool 2   [     1    ][     2    ][     3    ][     4    ][     5    ][
        6    ]
// Pool 3   [        1       ][        2       ][        3       ][
          4       ]
// Pool 4   [           1          ][           2          ][           3
             ]
//                                ^                          ^
//                                |                          |
//                              stake                    unstake
```

The minimal lock duration is 28 days so if a user stakes for just 28 days & 1 minute and unstakes in the points in time shown above, he will be able to claim the rewards for cycles of pools he did not wait the full duration or stayed in such cycles for just a couple of seconds.

In this specific example the user will be able to claim the rewards for:

- Pool 1: 4 - 9 (where 4 and 9 he stays staked for some seconds)
- Pool 2: 2 - 5 (where 2 and 5 he stays staked for some seconds)
- Pool 3: 2 - 3
- Pool 4: 1 - 3 (where 1 and 3 he stays staked for some seconds)

This is possible because when a user stakes or unstakes, the `cycleStart` and `cycleEnd` are set to the current pool cycle, regardless of the cycle's progress. The user can then call `claimRewardsToOwed()` for all cycles that have already passed and for cycles like Pool 4 cycle 3 he can return when cycle 4 starts and take his share of all rewards that were deposited in the duration of cycle 3 after he already unstaked.

Impact

Users can claim rewards for cycles they stayed for just a couple of seconds.

Recommendation

Consider implementing logic that will distribute rewards in a more fair manner.

Team Response

Acknowledged.

[M-01] The `onERC721Received()` Is Not Implemented Correctly

Severity

Medium Risk

Description

The `PositionManager.onERC721Received()` function intends to accept liquidity positions sent only by the admin to the `PositionManager` contract. It performs some validations regarding the liquidity position type that is being transferred and records the pair so other liquidity positions in the same pair are not accepted.

Location of Affected Code

File: PositionManager.sol

```
function onERC721Received(
      /* operator */
      address,
      address from,
      uint256 tokenId,
      /* data */
      bytes calldata
  ) external override nonReentrant returns (bytes4) {
@>  require(hasRole(DEFAULT_ADMIN_ROLE, from), "Transfers accepted only
   from DEFAULT_ADMIN_ROLE address");

    UniV3Nft memory nft = _getNftUniV3(tokenId);
    _validateNftUniV3(nft);

    // Check position NFT is full range
    require(_isFullRangeNftUniV3(nft), "Position is not full range"); //
       @audit not checked

    // Check if there already exists a position with the same token0 and
       token1 addresses
    require(_uniV3NftByToken0Token1[nft.token0][nft.token1] == 0, "
       Position already exists for token pair");

    // Check if for both tokens there is a swap router config (unless it
       is the $VOLT token)
    if (nft.token0 != address(voltToken)) {
        require(
            _voltSwapRouterConfigs[nft.token0].routerAddress != address
               (0), "Swap router config not set for token0"
        );
    }

    if (nft.token1 != address(voltToken)) {
        require(
            _voltSwapRouterConfigs[nft.token1].routerAddress != address
               (0), "Swap router config not set for token1"
        );
    }

    // Store the position
    _uniV3NftByToken0Token1[nft.token0][nft.token1] = tokenId;
    _uniV3NftIds.push(tokenId);

    return IERC721Receiver.onERC721Received.selector;
}
```

Impact

An attacker can exploit the `safeTransferFrom()` function by front-running the admin's transaction and invoking the `PositionManager.onERC721Received()` function with crafted arguments.

Specifically, the attacker can set the `from` parameter to the admin's address to bypass the `require(hasRole(DEFAULT_ADMIN_ROLE, from), "...")` check, which validates the sender's role.

By specifying a `tokenId` corresponding to the same pair the admin is attempting to transfer—one that the admin does not control—the attacker causes the admin's `safeTransferFrom()` call to revert. This effectively prevents the admin from transferring a liquidity position for the targeted pair, creating a denial-of-service scenario for this specific operation.

The admin, and indeed anyone, can still send liquidity positions to the `PositionManager` using the `transferFrom()` function instead of `safeTransferFrom()` . However, doing so will result in the `_uniV3NftByToken0Token1` and `_uniV3NftIds` mappings not being updated, rendering them inaccurate and effectively useless.

Recommendation

To limit liquidity positions the `PositionManager` interacts with, remove the `onERC721Received()` override and implement an admin-only deposit function. This function can call `safeTransferFrom()` , perform necessary checks, and record deposited `tokenIds` for verification in other functions.

Currently, functions like `addLiquidity()` , `collectFees()` , and `collectFeesOnly()` accept `tokenIds` transferred by anyone, which can be secured by using this deposit mechanism.

Team Response

Fixed.

[M-02] Staking and Reward Token Overlap Can Risk User Stakes

Severity

Medium Risk

Description

The `StakingVault.acceptPositionManagerRewards()` can be called by anybody and uses transferFrom with an arbitrary `from` value. This means that an attacker can use this function and transfer reward tokens from anybody that has given allowance to the `StakingVault` .

This is problematic since if the staking and the reward token are both $VOLT then users will first approve the `StakingVault` and then they will call `stake()` .

An attacker can front-run the stake call of the user and use this approved amount as a reward via the `acceptPositionManagerRewards()` .

Location of Affected Code

File: StakingVault.sol

```
function acceptPositionManagerRewards(address manager, uint256
   allowedAmount) external returns (uint256 amount) {
    endCycleIfNeeded();

    uint256 amountPerPool = allowedAmount / REWARD_POOL_COUNT;
    require(amountPerPool > 0, "Invalid amount"); // FIXME: either revert
       , or just ignore and do nothing

    // This can be different from original `amount` due to division
       rounding
    amount = amountPerPool * REWARD_POOL_COUNT;

@>  bool success = rewardsToken.transferFrom(manager, address(this),
   amount);
    require(success, "Transfer failed");

    for (uint256 i; i < REWARD_POOL_COUNT; i++) {
        uint256 poolId = _rewardPools[i].id;
        uint256 cycleId = _rewardPools[i].currentCycleId;

        _rewardPoolRewards[poolId][cycleId] += amountPerPool;
    }
}
```

Impact

A user could lose their tokens when they attempt to stake.

Recommendation

Always use `transferFrom` with a `msg.sender` as the `from` argument.

Team Response

Fixed.

[L-01] The `increaseLiquidity()` in `PositionManager` is Missing Slippage Protection

Severity

Low Risk

Description

The `PositionManager.addLiquidity()` provides a way for the admin to increase the liquidity of a position. The function uses `_addLiquidityUniV3()` that hardcodes the `amount0Min` and `amount1Min` arguments to 0 and exposes the admin to potential slippage.

Location of Affected Code

File: PositionManager.sol

```
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 check how increaseLiquidity works
}
```

Impact

Worst case scenario admin will provide less liquidity than expected and the excess tokens will be returned to the `PositionManager.sol`. From this point on these excess tokens cannot be used for anything except to be converted to Volt tokens and sent to the staking contract to be distributed as rewards.

Recommendation

Provide slippage protection for the `increaseLiquidity()`.

Team Response

Acknowledged.

[L-02] The `StakingVault` Does Not Support Fee-On-Transfer Tokens (or Weird Tokens)

Severity

Low Risk

Description

The `StakingVault` does not support fee-on-transfer tokens because the `stake()` function records the amount specified as an argument in `_userStakeInfo[user]`, rather than the actual amount of tokens received.

This will inflate the bonus of shares that the user will get and the last couple of users (or the last user depending on the number of users and the amounts they have staked) will not be able to unstake since the StakingVault will have less balance of staking token than the recorded amount in storage.

Location of Affected Code

File: StakingVault.sol

```
function _stake(address user, uint256 amount, uint16 periodInDays)
   internal returns (StakeInfo memory) {
function _restake(address user, uint256 amountToAdd, uint16 periodInDays)
    internal returns (StakeInfo memory) {
function acceptPositionManagerRewards(address manager, uint256
   allowedAmount) external returns (uint256 amount) {
```

Impact

Some users will experience loss of funds and bonus shares will be a bit inflated.

Recommendation

Record the token balance `StakingVault` before and after the transfer of the tokens to determine the actual received amount.

In general, before setting the staking and the reward tokens for the contract check whether these tokens have any weird functionalities such as reentrancy, fee on transfer, rebasing, reverting on transfer or approval and etc. Here is a list for a basic reference.

Team Response

Acknowledged.

[I-01] Outstanding Allowances and Max Approvals Could Cause Loss of Funds in Multiple Scenarios

Severity

Informational

Description

`PositionManager` uses `_setAllowanceMaxIfNeeded()` in multiple places. If the allowances towards the staking contract or a router are not enough, this function will max out the allowance.

Location of Affected Code

File: PositionManager.sol

```
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); // @audit not all tokens
        allow that I think
}
```

Impact

This is considered bad practice because if any approved contract is compromised or a vulnerability is discovered, the `PositionManager`'s tokens could be at risk of being stolen.

Furthermore, some tokens such as `UNI` and `COMP` will revert if you attempt to approve them for more than `type(uint96).max` which will revert the swaps to $VOLT or the `increaseLiquidity()` function for these tokens.

The tokens will be effectively blocked inside the `PositionManager` forever.

Recommendation

Consider approving entities only for the exact amount needed. This approach minimizes outstanding allowances, reducing the risk while still allowing necessary transactions.

If you plan to reset allowances to 0 after the main operation, be aware that certain tokens, like BNB, may revert when doing so.

Team Response

Fixed.

[I-02] The `userShares()` Does Not Show the Shares of the User Until the User Unstakes

Severity

Informational

Description

Users are not able to see how many shares they have in a pool until they unstakes.

Location of Affected Code

File: StakingVault.sol

```
function userShares(address user, uint256 poolId, uint256 cycleId)
   external view returns (uint256) {
    ClaimableReward memory r = _userRewardPoolClaimableReward[user][
       poolId];
    if (r.cycleStart == 0) return 0;
    if (r.cycleStart > cycleId) return 0;
@>  if (r.cycleEnd < cycleId) return 0;
    return r.shares;
}
```

Recommendation

Consider applying the following changes:

```
- if (r.cycleEnd < cycleId) return 0;
+ if (cycleId > currentPoolCycleId) return 0;
+ if (r.cycleEnd != 0 && r.cycleEnd < cycleId) return 0;
```

Team Response

Fixed.

[I-03] Use `safeERC20` in Order to Support Tokens Such as USDT

Severity

Informational

Description

USDT does not return a boolean on transfer which makes `StakingVault` revert on all transfer operations with it.

Location of Affected Code

All transfers in `StakingVault`

Impact

USDT cannot be used as a staking or reward token resulting in DOS.

Recommendation

Consider implementing `SafeERC20` if you want to support such weird tokens.

Team Response

Acknowledged.

[I-04] Typography

Severity

Informational

Description

- Use custom errors with if statements instead of require with string errors to save gas
- Use fixed solidity versions
- Remove console and `TransferHelper` imports in `PositionManager.sol` as they are not used
- Add NatSpec comments to all functions
- No need to use return in `_swapToVolt()`
- Emit events on every state-changing function

Recommendation

Consider implementing the recommendations above.

Team Response

Fixed.

[I-05] The `collectFees()` in `PositionManager.sol` Might Revert

Severity

Informational

Description

Fees in `UniV3` are gathered from the input token of a swap. There might be cases where for a period of time only swaps from token A to token B were made. This would mean that amount1 in this specific scenario is going to be 0 which will make `_swapToVolt()` revert and force the admin to use `collectFeesOnly()` instead.

Location of Affected Code

File: PositionManager.sol

```
function collectFees(uint256[] calldata tokenIds) external {
    require(tokenIds.length <= MAX_COLLECT_FEES_BATCH_SIZE, "tokenIds
       array too long");

    for (uint256 i = 0; i < tokenIds.length; i++) {
        uint256 tokenId = tokenIds[i];

        (uint256 amount0, uint256 amount1) = _collectFeesUniV3(tokenId);
        UniV3Nft memory nft = _getNftUniV3(tokenId);

@>      if (nft.token0 != address(voltToken)) {
            _swapToVolt(nft.token0, amount0);
        }

@>      if (nft.token1 != address(voltToken)) {
            _swapToVolt(nft.token1, amount1);
        }
    }

    _transferVoltToStakingVault();
}
```

Impact

The `collectFees()` in `PositionManager` might revert when only one of the tokens has accrued fees.

Recommendation

Consider applying the following changes:

```diff
-   if (nft.token0 != address(voltToken)) {
+   if (nft.token0 != address(voltToken) && amount0 != 0) {
    _swapToVolt(nft.token0, amount0);
}

-   if (nft.token1 != address(voltToken)) {
+   if (nft.token1 != address(voltToken) && amount1 != 0) {
    _swapToVolt(nft.token1, amount1);
}
```

Team Response

Fixed.

our shielding · Your smart contracts, our shielding · Your smart c

shieldify

Thank you!