SHERLOCK

Security Review For Highway

Collaborative Audit Prepared For:	**Highway**
Lead Security Expert(s):	**TessKimy**
Date Audited:	**July 29 - August 5, 2025**
Final Commit:	**aa606ce**

Introduction

Highway is a cross-chain liquidity migration protocol that enables seamless "lift-and-shift" transitions across DeFi ecosystems.

Designed for both users and protocols, Highway simplifies the movement of capital from one environment to another, whether it's between DEXs, lending markets, yield platforms, or token pools. It automates all the technical steps required from unwrapping LP positions, performing token swaps if necessary, bridging assets cross-chain, and re-deploying liquidity into new destinations. This unlocks a smoother way to migrate from underperforming protocols or chains and reallocates liquidity where incentives, infrastructure, or alignment are stronger.

Scope

Repository: aegas-io/sc-highway

Audited Commit: 86ca7c82331cf515dcf1381135a1e91b8c7cd663

Final Commit: aa606cecce49cf053284544427b270ad5671163c

Files:

- src/DexMigrateRouterUpgradeable.sol
- src/libraries/Errors.sol
- src/libraries/Routes.sol
- src/libraries/Structs.sol
- src/libraries/UniswapV2Util.sol
- src/libraries/UniswapV3Util.sol

Final Commit Hash

aa606cecce49cf053284544427b270ad5671163c

Findings

Each issue has an assigned severity:

- Medium issues are security vulnerabilities that may not be directly exploitable or may require certain conditions in order to be exploited. All major issues should be addressed.
- High issues are directly exploitable security vulnerabilities that need to be fixed.
- Low/Info issues are non-exploitable, informational findings that do not pose a security risk or impact the system's integrity. These issues are typically cosmetic or

related to compliance requirements, and are not considered a priority for remediation.

Issues Found

High	Medium	Low/Info
0	1	3

Issues Not Fixed and Not Acknowledged

High	Medium	Low/Info
0	0	0

Issue M-1: DoS in some edgecase Aerodrome V3 related actions

Summary

Aerodrome V3's NFT position manager is a little bit different than other uniswap/pancakeswap NFT position manager's implementations. There are some edgecase scenarios that interactions with Aerodrome's NFT position manager may revert.

Vulnerability Detail

In Aerodrome V3's NFT position manager has `refundETH` feature additional to other platforms. This function is called in `mint` and `increaseLiquidity` functions.

Basicly, if position manager has ETH higher than zero it refunds it to caller. The problem is we don't have any `receive()` implementation to accept that refund and it will revert in this case.

However, position manager doesn't accept direct ETH transfers itself, it only allows ETH from WETH source. There is a tricky way to trigger this DoS. Interestingly, Aerodrome V3 position manager's decrease liquidity function has `payable` state and it also doesn't use refund feature in this function which means we can leave position manager with higher than zero ETH after tx and it will revert if the next tx is Highway's position migration because it will try to `refundETH` to migration contract in mint function.

Impact

It's a very edgecase DoS attack. It can happen naturally without any external actor. It just need a sufficient condition to occur before migration call.

Code Snippet

https://github.com/aerodrome-finance/slipstream/blob/5b529b4d418a6d2e394391a153dfbd0c98de937d/contracts/periphery/NonfungiblePositionManager.sol#L195-L200

Tool Used

Manual Review

Recommendation

Consider adding receive() handler in order to handle this edgecase.

Discussion

b-hrytsak

Hello. Thank you for your submission.

It's an interesting one. And indeed, aerodrome can have a balance, and will return it during migration to a migration contract that is not ready for it.

This is potentially a DoS vector, although the price of such a transaction is not cheap for the caller, and the potential balance could become a target for MEV. Although this is an edge case for simple cases, it could also become a direct DoS attack vector on big migration tx, with dust amount of eth.

Fixed in commit: https://github.com/aegas-io/sc-highway/pull/2

Added an empty `receive()` function to handle these cases.

Thank you

Issue L-1: Incorrect error emit in case of unauthorized actions

Summary

In V3 migration cases, if NFT is not owned by the caller it reverts with following error:

`NotApprovedOrOwnerNftPosition()`

This is not 100% correct error message because approved caller's tx is already rejected. We check only for owner of the NFT and it reverts in case of an approved operator to migrate position.

Impact

It creates a discrepancy between error message and actual problem.

Code Snippet

```
require(
    IERC721(positionManagerSource).ownerOf(inParams_.tokenId) == _msgSender(),
    ↪  NotApprovedOrOwnerNftPosition()
);
```

Recommendation

Consider changing the error or consider adding operator check for allowed addresses.

Discussion

b-hrytsak

Hello, thank you for your submission.

There is indeed an inaccuracy, and the error indicates two types of access when in fact only one is valid.

Fixed in the commit: https://github.com/aegas-io/sc-highway/pull/1

Renamed (and description) error `NotApprovedOrOwnerNftPosition` to `NotNftPositionOwner`.

Thank you

Issue L-2: Missing update of `poolToRoute` and `poolToMellowLpWrapper` mapping in case of removal

This issue has been acknowledged by the team but won't be fixed at this time.

Summary

When we add a new pool for whitelisting, we update `poolToRoute` and `poolToMellowLpWrapper` mappings, however there is no removal action in case of we remove those pools from whitelisted address set.

Impact

It will cause out-dated contract states and especially for `getPoolRoute` function it will return out-dated data. It may cause problems in integrations.

Code Snippet

```
function revokePoolWhitelistBatch(WhitelistPoolParams[] calldata params_)
    external
    override
    onlyRole(ROUTES_MANAGER)
{
    require(params_.length != 0, EmptyArray());
    DexMigrateRouterStorage storage $ = _getDexMigrateRouterStorage();
    for (uint256 index; index < params_.length; index++) {
        WhitelistPoolParams calldata param = params_[index];
        if (RouteUtil.isMellow(param.route)) {
            require($.mellowPoolWhitelist.remove(param.pool), NotWhitelistedPool());
        } else {
            require($.generalWhitelist.remove(param.pool), NotWhitelistedPool());
            if (RouteUtil.isDexV2(param.route)) {
                $.v2WhitelistedPools.remove(param.pool);
            }
        }
        emit RevokePoolWhitelist(param.route, param.pool);
    }
}
```

Recommendation

Consider updating given mapping storage variables.

Discussion

b-hrytsak

Hi, thank you for your submission.

This submission can be considered as Info/Recommendation since it complies only with `Best Practices to Avoid Storage Bloat`.

The function will never return outdated data because, in reality, the created pool address will always belong to a specific type of dex (factory), and ‘it cannot migrate’. Therefore, `getPoolRoute` will always return real data in practice.

This will not cause integration problems, as there are no 'right cases' where it could lead to problems. Validation of whitelisting pool by `getPoolRoute` is invalid approach by default, as there is an `isWhitelist` method and need also handle cases with mellow.

With regard to gas, this is also questionable, and in practice there is no practical benefit from this.

Therefore, the only impact is to follow best practices to avoid bloating the chain, although the presence of this value in it can also bring certain benefits, such as checking “whether the pool has ever been whitelsited” or finding out the factory of a past whitelisted pool.

Will not be fixed.

Thank you.

Issue L-3: User's already gathered fee from position is used as input token

Summary

Current implementation, burns position liquidity and collect received token amounts. However, already gathered position fees are also included in this case because it doesn't collect them before.

Impact

It may cause unintentional position migrations especially if user doesn't know how much fee he's gathered from his position. User should decide a liquidity slippage for safety and he should account unclaimed yield into it.

User also pays protocol fee for this gathered yield.

Code Snippet

```
function _burnV2LiquidityAndPayFee(BurnV2LiquidityParams calldata params_)
    internal
    returns (address token0, address token1, uint256 amount0, uint256 amount1,
    ↪  uint256 fee0, uint256 fee1)
{
    IERC20(params_.source).safeTransferFrom(_msgSender(), params_.source,
    ↪  params_.liquidity);
    IBaseUniswapV2Pair(params_.source).burn(address(this));
    (token0, token1) = UniswapV2Util.getTokens(params_.source);
    (amount0, amount1) = _balanceOf(token0, token1, address(this));

    (fee0, fee1, amount0, amount1) = _processFee(token0, token1, params_.maxFee,
    ↪  amount0, amount1);
}
```

```
function decreaseAndCollectLiquidity(
    address positionManager_,
    uint256 tokenId_,
    uint128 liquidity_,
    address recipient_,
    bool burnNftPositionAfter
) internal returns (uint256 amount0, uint256 amount1) {
    INonfungiblePositionManager positionManager =
    ↪  INonfungiblePositionManager(positionManager_);

    positionManager.decreaseLiquidity(
```

```
        INonfungiblePositionManager.DecreaseLiquidityParams({
            tokenId: tokenId_,
            liquidity: liquidity_,
            amount0Min: 0,
            amount1Min: 0,
            deadline: block.timestamp
        })
    );

    (amount0, amount1) = positionManager.collect(
        INonfungiblePositionManager.CollectParams({
            tokenId: tokenId_,
            recipient: recipient_,
            amount0Max: type(uint128).max,
            amount1Max: type(uint128).max
        })
    );

    if (burnNftPositionAfter) {
        positionManager.burn(tokenId_);
    }
}
```

Recommendation

Consider transferring gathered yield to user before starting migration or add a user input and let him to decide to use it or not.

Discussion

b-hrytsak

Hello, and thanks for the submission.

By design, our migrator treats a full-position migration as migrating both liquidity **and any accrued, unclaimed fees**. This mirrors the idea of a “complete position.” In such flows, including fees is intentional so users don’t lose or forget accrued yield post-migration and don’t need extra steps to claim it later.

For users who prefer not to migrate fees, this should be handled by the FE/off-chain flow: the UI should prompt users to claim/withdraw fees before migration or clearly show the fees that will be included if they proceed.

To account for all possible edge cases, these fees are “claimed.” Depending on the DEX, some do not store fees within the position and therefore cannot claim them from migrator contract (aerodrome uniswap v2), while others directly integrate them into the liquidity, making it impossible to claim them separately (uniswap v2).

Note: The v2 code snippet mentioned is not specific to this problem. v2 not have this issue.

But looking at your submission, there is a problem with partial transfers where we transfer a part liquidity, but all fees claim always, so for the better flow, need to give a choice of how many fees are claimed on the migration.

Fixed in: https://github.com/aegas-io/sc-highway/pull/3

Thank you.

Disclaimers

Sherlock does not provide guarantees nor warranties relating to the security of the project.

Usage of all smart contract software is at the respective users' sole risk and is the users' responsibility.