Pashov Audit Group

Funnel Security Review

August 27th 2025 - August 29th 2025

Contents

1. About Pashov Audit Group

Pashov Audit Group consists of 40+ freelance security researchers, who are well proven in the space - most have earned over $100k in public contest rewards, are multi-time champions or have truly excelled in audits with us. We only work with proven and motivated talent.

With over 300 security audits completed — uncovering and helping patch thousands of vulnerabilities — the group strives to create the absolute very best audit journey possible. While 100% security is never possible to guarantee, we do guarantee you our team's best efforts for your project.

Check out our previous work here or reach out on Twitter @pashovkrum.

2. 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.

3. Risk Classification

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

Impact

- **High** - leads to a significant material loss of assets in the protocol or significantly harms a group of users
- **Medium** - leads to a moderate material loss of assets in the protocol or moderately harms a group of users
- **Low** - leads to a minor material loss of assets in the protocol or harms a small group of users

Likelihood

- **High** - attack path is possible with reasonable assumptions that mimic on-chain conditions, and the cost of the attack is relatively low compared to the amount of funds that can be stolen or lost
- **Medium** - only a conditionally incentivized attack vector, but still relatively likely
- **Low** - has too many or too unlikely assumptions or requires a significant stake by the attacker with little or no incentive

4. About Funnel

Funnel is a system for deploying and managing Uniswap V3–style liquidity pools on Hyperliquid, enabling fee collection, conversion, and bridging between HyperEVM and HyperCore. It automates HIP-1 ticker acquisition, token burns, and creator fee claims through a vault-based architecture with configurable fee management.

5. Executive Summary

A time-boxed security review of the **acc8-labs/funnel-vault** and **acc8-labs/v3-core** repositories was done by Pashov Audit Group, during which **Shaka**, **0xTheBlackPanther**, **imsrybr0** engaged to review **Funnel**. A total of **14** issues were uncovered.

Protocol Summary	
Project Name	Funnel
Protocol Type	DEX
Timeline	August 27th 2025 - August 29th 2025

Review commit hashes:

- 95019239bb194f7af3bf27fd91b1238efe955af4 (acc8-labs/funnel-vault)
- 2c14fb74cc761f51a33971403ace3453abe2b554 (acc8-labs/v3-core)

Fixes review commit hash:

- 14774a70ffeaad0108d8a08ea3a7018cfa5178bf (acc8-labs/funnel-vault)

Scope

`UniswapV3Factory.sol` `UniswapV3Pool.sol` `IUniswapV3Factory.sol` `FunnelVaultUpgradeable.sol`

6. Findings

Findings count

Severity	Amount
High	2
Low	12
Total findings	**14**

Summary of findings

ID	Title	Severity	Status
[H-01]	`FunnelVaultUpgradeable` cannot send transactions to HyperCore	High	Resolved
[H-02]	swapTokens does not account for SwapX router fees when the out token is native	High	Resolved
[L-01]	Deadline not checked for swaps through V2	Low	Resolved
[L-02]	Emergency withdrawal cannot recover `HYPE`	Low	Resolved
[L-03]	Swap reverts when `tokenIn` is `WETH`	Low	Resolved
[L-04]	Undistributed dust of fees due to rounding errors	Low	Resolved
[L-05]	Missing `_disableInitializers()` call in constructor	Low	Resolved
[L-06]	Wrong `to` address	Low	Resolved
[L-07]	Ineffective token burning mechanism in fee handling	Low	Resolved
[L-08]	Missing setter for `protocolFeeClaimer`	Low	Resolved
[L-09]	Uninitialized fee splits cause loss of fee accounting	Low	Resolved
[L-10]	burnToken does not verify token transfer success	Low	Resolved
[L-11]	FunnelVault will not be able to spend all donations/protocol fees	Low	Acknowledged
[L-12]	`withdrawNative` and `depositNative` cannot be used for WHYPE/HYPE	Low	Resolved

High findings

[H-01] `FunnelVaultUpgradeable` cannot send transactions to HyperCore

Severity

Impact: Medium

Likelihood: High

Description

`sendNativeTokenToLayer1()` bridges `HYPE` from HyperEVM to HyperCore. These funds are credited to the vault contract address in HyperCore, so it is required that the contract implements the logic to send transactions from the HyperEVM to HyperCore.

As `FunnelVaultUpgradeable` does not implement a mechanism to send transactions to HyperCore, the funds will be locked in HyperCore until the contract is upgraded, adding the required logic.

Recommendations

Implement the logic to send transactions from the HyperEVM to HyperCore following the Hyperliquid documentation.

[H-02] swapTokens does not account for SwapX router fees when the out token is native

Severity

Impact: Medium

Likelihood: High

Description

When the out token is native, the `SwapX` router charges fees on the out amount of the swap in both `swapV2ExactIn` and `swapV3ExactIn`, sending the remainder to the recipient but still returning the original out amount to the caller.

SwapX@takeFee

```
    function takeFee(address tokenIn, uint256 amountIn) internal returns (uint256){
        if (feeExcludeList[msg.sender])
            return 0;

        uint256 fee = amountIn.mul(feeRate).div(feeDenominator);

        if ( tokenIn == address(0) || tokenIn == WETH ) {
            require(address(this).balance > fee, "insufficient funds");
            (bool success, ) = address(feeCollector).call{ value: fee }("");
            require(success, "SwapX: take fee error");
        } else
            IERC20Upgradeable(tokenIn).safeTransferFrom(msg.sender, feeCollector, fee);

        emit FeeCollected(tokenIn, msg.sender, fee, amountIn, block.timestamp);

        return fee;
    }
```

SwapX@swapV2ExactIn

```
    function swapV2ExactIn(
        address tokenIn,
        address tokenOut,
        uint256 amountIn,
        uint256 amountOutMin,
        address poolAddress
    ) payable public nonReentrant whenNotPaused returns (uint amountOut){
        // ...

        bool nativeOut = false;
        if (tokenOut == address(0))
            nativeOut = true; <@

        // ...

        if (nativeOut) {
            amountOut = IERC20Upgradeable(WETH).balanceOf(address(this)).sub(balanceBefore); <@
            IWETH(WETH).withdraw(amountOut);
            uint256 fee = takeFee(address(0), amountOut); <@
            (bool success, ) = address(msg.sender).call{value: amountOut-fee}(""); <@
            require(success, "SwapX: send ETH out error");
        } else {
            amountOut = IERC20Upgradeable(tokenOut).balanceOf(msg.sender).sub(balanceBefore);
        }
        require(
            amountOut >= amountOutMin,
            'SwapX: insufficient output amount'
        );
    }
```

SwapX@swapV3ExactIn

```
    function swapV3ExactIn (
            ExactInputSingleParams memory params
    ) external payable nonReentrant whenNotPaused checkDeadline(params.deadline) returns
(uint256 amountOut) {
        // ...
```

```
        bool nativeOut = false;
        if (params.tokenOut == WETH)
            nativeOut = true;

        // ...

        amountOut = exactInputInternal( <@
            params.amountIn,
            nativeOut ? address(0) : params.recipient,
            params.sqrtPriceLimitX96,
            SwapCallbackData({path: abi.encodePacked(params.tokenIn, params.fee,
params.tokenOut), payer: msg.sender, payerOrigin: msg.sender})
        );

        require(amountOut >= params.amountOutMinimum, "SwapX: insufficient out amount");

        if (nativeOut) {
            IWETH(WETH).withdraw(amountOut);
            uint256 fee = takeFee(address(0), amountOut); <@
            (bool success, ) = address(params.recipient).call{value: amountOut-fee}(""); <@
            require(success, "SwapX: send ETH out error");
        }
    }
```

This leads to the `FunnelVault` accounting for more tokens than actually received from the swap.

FunnelVaultUpgradeable@swapTokens

```
    function swapTokens(SwapParams calldata swapParams, address swapRouter, bool isBurn)
external onlyRole(EXECUTER_ROLE) nonReentrant returns (uint256 amountOut) {
        // ...

        if (swapParams.useV3) {
            ISwapX.ExactInputSingleParams memory params = ISwapX.ExactInputSingleParams({
                tokenIn: swapParams.tokenIn,
                tokenOut: swapParams.tokenOut,
                fee: swapParams.fee,
                recipient: address(this),
                deadline: swapParams.deadline,
                amountIn: swapParams.amountIn,
                amountOutMinimum: swapParams.amountOutMin,
                sqrtPriceLimitX96: 0
            });

            if (params.tokenIn == address(0)) {
                amountOut = ISwapX(swapRouter).swapV3ExactIn{value: params.amountIn}(params);
            } else {
                amountOut = ISwapX(swapRouter).swapV3ExactIn(params); <@
            }
        } else {
            if (swapParams.poolAddress == address(0)) {
                revert ZeroAddress();
            }

            if (swapParams.tokenIn == address(0)) {
```

```
                amountOut = ISwapX(swapRouter).swapV2ExactIn{value: swapParams.amountIn}(
                    swapParams.tokenIn,
                    swapParams.tokenOut,
                    swapParams.amountIn,
                    swapParams.amountOutMin,
                    swapParams.poolAddress
                );
            } else {
                amountOut = ISwapX(swapRouter).swapV2ExactIn( <@
                    swapParams.tokenIn,
                    swapParams.tokenOut,
                    swapParams.amountIn,
                    swapParams.amountOutMin,
                    swapParams.poolAddress
                );
            }
        }

        if (amountOut == 0) {
            revert SwapFailed();
        }

        if (swapParams.tokenOut == WETH && swapParams.useV3) {
            if(isBurn){
                addToPoolBurn(address(0), swapParams.payingPool, amountOut); <@
            } else {
                addToPoolHIP(address(0), swapParams.payingPool, amountOut); <@
            }
        } else {
            if(isBurn){
                addToPoolBurn(swapParams.tokenOut, swapParams.payingPool, amountOut); <@
            } else {
                addToPoolHIP(swapParams.tokenOut, swapParams.payingPool, amountOut); <@
            }
        }

        emit TokenSwapped(swapParams.tokenIn, swapParams.tokenOut, swapParams.amountIn,
amountOut, swapRouter);
    }
```

Recommendations

- On Funnel controlled `SwapX` routers, add the `FunnelVault` to the `feeExcludeList` or set the `feeRate` to 0.
- To handle both controlled and non controlled `SwapX` routers, do not rely on the returned out amount and instead compare balance before and after the swap (similar to Fee on Transfer tokens).

Low findings

[L-01] Deadline not checked for swaps through V2

In the `swapTokens()` function, when `swapParams.useV3` is false, `ISwapX.swapV2ExactIn()` is called. This function does not check if `swapParams.deadline` has passed, which can provoke that the swap being executed later than expected at a worse price.

It is recommended to add the deadline check in the `swapTokens()` function when the swap uses V2.

```
        } else {
            if (swapParams.poolAddress == address(0)) {
                revert ZeroAddress();
            }

+           if (block.timestamp > swapParams.deadline) {
+               revert TransactionTooOld();
+           }
```

[L-02] Emergency withdrawal cannot recover `HYPE`

`emergencyWithdrawToken()` is meant to be called in the case of an emergency to withdraw tokens from the contract. However, it does not implement any logic to recover native tokens.

It is recommended to implement a mechanism to recover native tokens in the `emergencyWithdrawToken()` function.

[L-03] Swap reverts when `tokenIn` is `WETH`

When `tokenIn` is `WETH`, `SwapX.swapV3ExactIn()` expects receiving native tokens.

```
File: SwapX.sol

        if (params.tokenIn == WETH || params.tokenIn == address(0)) {
            params.tokenIn = WETH;
            require(msg.value >= params.amountIn, "SwapX: amount in and value mismatch");
```

However, in `FunnelVaultUpgradeable.swapTokens()`, when `tokenIn` is `WETH`, no native tokens are sent in the call to `swapV3ExactIn()` and `WETH` is instead approved for `SwapX`, which will cause the transaction to revert.

While this limitation can be circumvented by calling `withdrawNative()` and using `address(0)` as `tokenIn`, it is unclear if this is the expected behavior. If this is the case, consider reverting early with an explicit error message when `tokenIn` is `WETH`. Otherwise, consider applying the following changes:

```
        if (swapParams.tokenIn == address(0)) {
            if (swapParams.amountIn > address(this).balance) {
                revert InsufficientBalance();
            }
        } else {
            if (swapParams.amountIn > IERC20(swapParams.tokenIn).balanceOf(address(this))) {
                revert InsufficientTokenBalance();
            }
-           IERC20(swapParams.tokenIn).forceApprove(swapRouter, swapParams.amountIn);
+           if (swapParams.tokenIn == WETH) {
+               IWETH(WETH).withdraw(swapParams.amountIn);
+           } else {
+               IERC20(swapParams.tokenIn).forceApprove(swapRouter, swapParams.amountIn);
+           }
        }

(...)

-           if (params.tokenIn == address(0)) {
+           if (params.tokenIn == address(0) || params.tokenIn == WETH) {
                amountOut = ISwapX(swapRouter).swapV3ExactIn{value: params.amountIn}(params);
```

[L-04] Undistributed dust of fees due to rounding errors

`splitFeesCollected()` splits the fees into protocol and utility fees according to the configured fee percentages. The resulting fees can leave undistributed dust amounts due to rounding errors in the fee calculations.

Depending on the valuation of the tokens and their decimal precision, the accumulation of these dust amounts could be significant over time.

Given that the percentage of the two fees is guaranteed to always add up to 100%, it is recommended to apply the following change to guarantee the distribution of all the fees:

```
    function splitFeesCollected(uint256 amount, FeeSplit memory feeSplit) internal view returns
(uint256 protocolFee, uint256 utilityFee) {
        protocolFee = (feeSplit.protocolFeePercentage * amount) / FEE_PERCENTAGE_PRECISION;
-       utilityFee = (feeSplit.utilityFeePercentage * amount) / FEE_PERCENTAGE_PRECISION;
+       utilityFee = amount - protocolFee;
    }
```

[L-05] Missing `_disableInitializers()` call in constructor

The constructor of `FunnelVaultUpgradeable` does not call `_disableInitializers()`, which could allow unintended initialization of the implementation contract.

It is recommended to call `_disableInitializers()` in the constructor.

[L-06] Wrong `to` address

In the `NativeTokenSentToLayer1` event emitted by the `sendNativeTokenToLayer1` function, the `to` parameter is set to the system address `0x2222222222222222222222222222222222222222`. However, the recipient of the funds in the L1 is `address(this)`.

Consider updating the event to reflect the actual recipient address, or remove the `to` parameter, as it will always have the same value.

[L-07] Ineffective token burning mechanism in fee handling

The bug occurs when attempting to "burn" tokens by transferring them to `address(0)` instead of properly reducing the total supply, which fails to achieve true burning and merely relocates inaccessible tokens while keeping the reported total supply inflated.

Transferring ERC20 tokens to `address(0)` (or any dead address) moves them to an inaccessible balance without calling a proper `burn` function, so `totalSupply` remains unchanged (e.g., 1,000 total stays 1,000 even after "burning" 100), unlike true burns that reduce it (to 900). This is not actual burning—tokens count toward supply metrics but are stuck.

The issue is in the internal `addToClaimer` function, specifically the line `IERC20(token).safeTransfer(address(0), amount);` within the `if(claimer == address(0))` conditional block, invoked from `claimProtocolFees`.

It triggers if `protocolFeeClaimer` or a pool's `claimer` is configured to `address(0)` (possible via future upgrades, admin setters, or logic changes), during `claimProtocolFees` when splitting fees, causing improper transfers instead of burns.

One clear impact of it will be enabling potential unintended recovery from `address(0)`, while native ETH (`token == address(0)`) silently fails without burning or reverting, leaving funds unaccounted.

This inconsistency with `burnToken`'s 0xdead address handling undermines tokenomics and risks supply integrity in fee management.

Recommendation is to refactor the zero-claimer branch to use a consistent dead address (e.g., 0x000...dEaD) for pseudo-burns, add explicit reversion for native ETH (`if (token == address(0)) revert;`), and prevent zero claimers via validations or immutable setters. If tokens support it, prefer calling `burn(amount)` for true supply reduction.

[L-08] Missing setter for `protocolFeeClaimer`

In the funnel vault contract `protocolFeeClaimer` is set only in the initialize to admin and has no setter function. If the admin address changes (e.g., role revoked/granted to a new address) or the claimer key is lost/compromised, protocol fees added via addToClaimer become unclaimable forever, as only the old claimer can call claimFees.

Admin can emergency withdraw, but that abandons accounting (known issue), risking over-claims or state corruption.

This leads to permanent lockup of protocol fees if claimer changes, affecting protocol sustainability.

This is how it can arise

- Init sets claimer=admin.
- Revoke DEFAULT_ADMIN_ROLE from initial admin, grant to new.
- Accrue protocol fees -> added to old claimer. New admin can't claim, fees locked.

Recommendation is to add `setProtocolFeeClaimer(address newClaimer)` restricted to `DEFAULT_ADMIN_ROLE`, with `newClaimer != address(0)` check.

[L-09] Uninitialized fee splits cause loss of fee accounting

The fee split configurations `poolFeeSplitsBeforeMigration` and `poolFeeSplitsAfterMigration` are never initialized in the `initialize()` function, defaulting to (0,0). When `claimProtocolFees()` executes, it calls `splitFeesCollected()` with these zero percentages:

```
(uint256 protocolFee, uint256 utilityFee) = splitFeesCollected(amount0,
poolFeeSplitsBeforeMigration);
// With (0,0) split: protocolFee = (0 * amount0) / 10000 = 0
//                   utilityFee = (0 * amount0) / 10000 = 0
```

The tokens are successfully collected from uniswap pools via `collectProtocol()`, but both `addToPoolHIP()` and `addToClaimer()` calls receive zero amounts, leaving collected fees untracked in the vault's balance.

This occurs every time `claimProtocolFees()` is called before the admin manually sets fee splits via `setPoolFeeSplits()`. The contract enforces that the new fee splits sum to 10,000, but doesn't prevent using the default (0,0) values.

This way **all collected protocol fees become permanently untracked in vault accounting, effectively vanishing from the protocol's fee distribution system until manual recovery.**

Recommendation

Initialize fee splits with sensible defaults during contract initialization.

```
function initialize(address admin, address executer) external initializer {
    // ... existing code ...

    // Initialize fee splits per comments: 25%/75% before, 50%/50% after
    poolFeeSplitsBeforeMigration = FeeSplit(2500, 7500);
    poolFeeSplitsAfterMigration = FeeSplit(5000, 5000);
}
```

This ensures fee accounting works correctly from deployment without requiring separate configuration transactions.

[L-10] burnToken does not verify token transfer success

The `burnToken()` function in `FunnelVaultUpgradeable` uses a direct ERC-20 transfer to the burn address but does not check the return value. If the token does not strictly follow ERC-20 standards (e.g a non-compliant token that returns false on failure, the vault would assume the burn succeeded even if it failed.

In such a case, the mapping is decremented and the event emitted, but the tokens remain in the contract. This results in token funds stuck in the vault (*since the vault thinks they were burned*) and potential accounting inconsistencies. For instance, a token like USDT (which doesn't revert on failure) could silently fail to transfer, leaving the vault holding tokens that are no longer tracked in burnFees.

Recommendation is to add a check if transfer is successful, and if not, revert the trx.

[L-11] FunnelVault will not be able to spend all donations/protocol fees

Every time the `FunnelVault` needs to swap donations/protocol fees, new protocol fees will be generated again, which the vault will need to claim and eventually swap too.

Consider adding a way to exempt the `FunnelVault` from paying swap fees.

[L-12] `withdrawNative` and `depositNative` cannot be used for WHYPE/HYPE

It is not possible to wrap/unwrap tokens from the Burn pool.

Consider adding a `isBurn` parameter to those function to allow that.