StarHeroes Launchpool Security Review

Duration: May 16, 2025 - May 24, 2025

July 30, 2025

Conducted by **KeySecurity**
Georgi Krastenov, Lead Security Researcher

Table of Contents

1 About KeySecurity

KeySecurity is an innovative Web3 security company that hires top talented security researchers for your project. We have conducted over 40 security reviews for different projects which hold over $500,000,000 in TVL. For security audit inquiries, you can reach out to us on Twitter/X or Telegram @gkrastenov, or check our previous work here.

2 About StarHeroes

StarHeroes is #1 Multiplayer Esports Space Shooter with $1.6M Decentralized Esports League.

3 Disclaimer

Audits are a time, resource, and expertise bound effort where trained experts evaluate smart contracts using a combination of automated and manual techniques to identify as many vulnerabilities as possible. Audits can show the presence of vulnerabilities **but not their absence**.

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** - leads to a significant loss of assets in the protocol or significantly harms a group of users.
- **Medium** - only a small amount of funds can be lost or a functionality of the protocol is affected.
- **Low** - any kind of unexpected behaviour that's not so critical.

4.2 Likelihood

- **High** - direct attack vector; the cost is relatively low to the amount of funds that can be lost.
- **Medium** - only conditionally incentivized attack vector, but still relatively likely.
- **Low** - too many or too unlikely assumptions; provides little or no incentive.

4.3 Actions required by severity level

- **Critical** - client **must** fix the issue.
- **High** - client **must** fix the issue.
- **Medium** - client **should** fix the issue.
- **Low** - client **could** fix the issue.

5 Executive summary

Overview

Project Name	StarHeroes Launchpool
Repository	https://github.com/starheroescommunity/sh-launchpool-contracts
Commit hash	8941fa7d42dd8df5594feb5bf3d7acbd44247de9
Review Commit hash	4c218b458d7e859a66f3fdc35adafc15e252e0c7
Documentation	N/A
Methods	Manual review

Scope

UniswapV3SwapExecutor.sol
UniswapV3LiquidityManager.sol
TokenPublicMintable.sol
StarHeroesMultiAssetLaunchpool.sol
ShStakingStToken.sol
ShStakingCore.sol
ShStaking.sol
MultiAssetLaunchPool.sol

Timeline

May 16, 2025	Audit kick-off
May 24, 2025	Mitigation review
July 30, 2025	Final report

Issues Found

Severity	**Count**
Critical	1
High	2
Medium	2
Low	1
Information	6
Total	**12**

6 Findings

6.1 Critical

6.1.1 Users can repeatedly call swapRewardAndStake draining contract rewards

Severity: *Critical*

Context: StarHeroesMultiAssetLaunchpool.sol

Description: Currently, users participating in the MultiAsset Launchpool can choose between two options to handle their earned rewards:

- Claim the reward via the `getReward` function
- Swap and stake the reward via the `swapRewardAndStake` function.

When a user claims their reward using `getReward`, their reward debt is updated to prevent repeated claims:

```
uint256 accumulated = (boosted * rewardPerPoint) / 1e18;
uint256 owed = accumulated - _userRewardDebt[msg.sender];
if (owed == 0) revert InsufficientRewards();
_userRewardDebt[msg.sender] = accumulated; // note
_rewardToken.safeTransfer(msg.sender, owed);
```

However, this mechanism is not applied in the `swapRewardAndStake` function. As a result, users can repeatedly call `swapRewardAndStake` to swap and stake their rewards multiple times—potentially draining the contract's entire balance.

Recommendation: Update `swapRewardAndStake` to properly update the user's reward debt after claiming, just like in `getReward`.

Resolution and Client comment: Resolved.

6.2 High

6.2.1 Users lose pending rewards if they unstake before claiming

Severity: *High*

Context: MultiAssetLaunchPool.sol#L193

Description: When a user starts staking their tokens in the `MultiAssetLaunchPool` contract, they earn rewards based on the ratio of their points to the total points of all staked tokens, the reward rate, and the duration staked.

To claim their rewards, the user needs to call the `getReward` function, which transfers the earned tokens to their address.

```
        uint256 boosted = _userEffectivePoints[msg.sender];
        uint256 accumulated = (boosted * rewardPerPoint) / 1e18;
        uint256 owed = accumulated - _userRewardDebt[msg.sender];
        if (owed == 0) revert InsufficientRewards();
        _userRewardDebt[msg.sender] = accumulated;
        _rewardToken.safeTransfer(msg.sender, owed);
```

A user can lose their earned rewards if they unstake before claiming. When the user unstakes, their effective points are reset because the `_updateEffectivePoints` function is called:

```
        stakeInfo.amount -= amount;
        stakeInfo.points -= points;
        userTotalPoints[msg.sender] -= points;

        _updateEffectivePoints(msg.sender);
```

The effective points participate in reward calculations: `accumulated = (boosted * rewardPerPoint )/ 1e18`; When effective points are reset (i.e., `boosted` = 0), the `accumulated` amount also becomes 0.

Recommendation: Add logic to automatically claim rewards before unstaking (e.g., auto-call `getReward` inside `unstake`).

Resolution and Client comment: Resolved.

6.2.2 User loses accumulated reward when making a second stake before calling getReward

Severity: *High*

Context: MultiAssetLaunchPool.sol#L194

Description: When a user claims their reward, the contract calculates their accumulated reward and subtracts their reward debt:

```
        uint256 accumulated = (boosted * rewardPerPoint) / 1e18;
        uint256 owed = accumulated - _userRewardDebt[msg.sender];
```

The difference (owed) represents the reward earned during staking. If a user makes a second stake before calling `getReward`, they will lose the accumulated reward from their previous staking period. Example:

- User stakes tokens.
- User waits 10 days.
- User makes a second stake.
- User calls `getReward` expecting to receive rewards for the 10 days.

This doesn't happen because:

- On staking, `_updateEffectivePoints` updates the user's reward debt:

```
_userRewardDebt[user] = (newEffective * rewardPerPoint) / 1e18;
```

- As a result, the `accumulated` value and `rewardDebt` are nearly equal.
- `owed` becomes 0 (or very small), and the user loses their reward from the first staking period.

Recommendation: Introduce a mechanism to harvest pending rewards before updating the user's reward debt on additional stakes.

Resolution and Client comment: Resolved.

6.3 Medium

6.3.1 Missing staking period check in stakeNFTs allows boosting points after end time

Severity: *Medium*

Context: MultiAssetLaunchPool.sol#L147

Description: Every user can boost their base points by staking NFTs. Each staked NFT increases the user's points by 0.5%. To stake an NFT, the user calls the `stakeNFTs` function.

Currently, the `stakeNFTs` function is missing the `whenStakingActive` modifier, which is supposed to prevent staking after the staking period ends (`block.timestamp > endTime`).

Due to this missing restriction, users can start or continue staking NFTs even after the staking period has ended, allowing them to increase their base points and earn additional (incorrect) rewards.

Recommendation: Add the `whenStakingActive` modifier to `stakeNFTs` to enforce the staking period restriction.

Resolution and Client comment: Resolved.

6.3.2 Rewards continue accruing after endTime

Severity: *Medium*

Context: MultiAssetLaunchPool.sol#L214

Description: When the contract is initialized, the rewardRate is calculated as:

```
rewardRate = totalReward / duration;
// where duration = endTime - startTime
```

During this `duration`, users can stake tokens or NFTs and earn rewards.

The reward earning period starts only after the first stake (since `lastUpdateTime = block.timestamp` is set on first stake). The `updateReward` function only checks: `if (totalPoints > 0)`. Rewards **do not stop** accruing when `block.timestamp > endTime`. This means users can stake near or even *after* `endTime`, and still continue earning rewards **until contract funds are drained**. Example:

- User A stakes at the beginning of the program.
- `endTime` is reached, but User A hasn't claimed yet.
- User B stakes just after `endTime`.
- User B can still earn rewards, potentially claiming what was meant for User A.

Recommendation: Enforce reward earning cutoff at `endTime`.

Resolution and Client comment: Resolved.

6.4 Low

6.4.1 User can claim rewards when the contract is paused

Severity: *Low*

Context: ShStakingCore.sol#L133

Description: Currently, a user can claim their reward even when the contract is paused. This happens because the whenNotPaused modifier is missing in the `claimRewards()` function.

Recommendation: Add the `whenNotPaused` modifier to the `claimRewards` function.

Resolution and Client comment: Acknowledged.

6.5 Information

6.5.1 Redundant events and errors

Severity: *Information*

Context: Global

Description: The following events and errors are redundant:

```
    // In ShStakingCore.sol
    event TokenPoolUpdated(uint256 newPoolSize);
    event UnboundTimeUpdated(uint256 daysNumber);
    event LockupTimeUpdated(uint256 daysNumber);

    // In UniswapV3SwapExecutor
    error InvalidPath();
    error PathNotConfigured();
    error Unauthorized();
    error QuoterCallFailed(string reason);
```

Recommendation: Remove the redundant events and errors.

Resolution and Client comment: Resolved.

6.5.2 Use RATE_DIVIDER instead of the hardcoded value

Severity: *Information*

Context: ShStakingCore.sol#L105

Description: Use the `RATE_DIVIDER` variable instead of the hardcoded value during the calculation of the reward.

```
        if (elapsedTime >= unbondingDuration) {
            availableToClaim = tokensToClaim;
        } else {
            uint256 stablePart = (tokensToClaim * unboundInitialPercent) / 100000;
                //@note use RATE_DIVIDER

            availableToClaim = stablePart + (elapsedTime * (tokensToClaim -
                stablePart)) / unbondingDuration;
```

Recommendation: Use the `RATE_DIVIDER` variable instead of 100000.

Resolution and Client comment: Resolved.

6.5.3 Unused code in UniswapV3SwapExecutor

Severity: *Information*

Context: UniswapV3SwapExecutor.sol

Description: The `EnumerableSet` in `UniswapV3SwapExecutor` is never used. Also, the addresses of the WETH and quoter contracts are not used.

Recommendation: Remove the redundant code.

Resolution and Client comment: Resolved.

6.5.4 Redundant console.log

Severity: *Information*

Context: StarHeroesMultiAssetLaunchpool.sol#L108-L112

Description: In the `StarHeroesMultiAssetLaunchpool` contract, when the contract's balance is checked, if it is lower than the user's reward, a console log displays the information, including the message, the actual balance and the reward. This console log is primarily used during the development of the contract and will not be used in production.

Recommendation: Remove the console log and the import.

Resolution and Client comment: Resolved.

6.5.5 Redundant swap payload input in swapRewardAndStake

Severity: *Information*

Context: StarHeroesMultiAssetLaunchpool.sol#L94

Description: Users can provide a swap execution payload as an input argument to the `swapRewardAndStake` function.

```
    function swapRewardAndStake(
        uint256 _minStarOut,
        address _stakingToken,
        bytes calldata _payload //@audit
    ) external payable {
```

This payload is then forwarded to the `ISwapExecutor` contract.

However, the `ISwapExecutor` contract does not use the `_payload` during the swap, which means this input argument is redundant.

Recommendation: Remove the unused `_payload` argument from `swapRewardAndStake`.

Resolution and Client comment: Acknowledged.

6.5.6 Redundant function

Severity: *Information*

Context: StarHeroesMultiAssetLaunchpool.sol#L156

Description: The`claimRewards` function in the `StarHeroesMultiAssetLaunchpool` contract is redundant.

```
    function claimRewards() external whenNotPaused {
        updateReward();

        uint256 boosted = _userEffectivePoints[msg.sender];
        uint256 accumulated = (boosted * rewardPerPoint) / 1e18;
        uint256 owed = accumulated - _userRewardDebt[msg.sender];

        if (owed == 0) revert InsufficientRewards();
        _userRewardDebt[msg.sender] = accumulated;
        _rewardToken.safeTransfer(msg.sender, owed);
        emit RewardPaid(msg.sender, owed);
    }
```

Recommendation: Remove the redundant function.

Resolution and Client comment: Resolved.